UNIVERSITY OF ILLINOIS
URBANA

AERONOMY REPORT
NO. 97

IMPLEMENTATION OF CONTEXT INDEPENDENT CODE ON A NEW ARRAY PROCESSOR: THE SUPER-65

by
R. O. Colbert
S. A. Bowhill

June 1, 1981

Library of Congress ISSN 0568-0581

Aeronomy Laboratory
Department of Electrical Engineering
University of Illinois
Urbana, Illinois

Supported by
National Science Foundation

UILU-ENG-81 2503

AERONOMY REPORT

NO. 97

IMPLEMENTATION OF CONTEXT INDEPENDENT CODE ON A NEW ARRAY PROCESSOR: THE SUPER-65

by

R. O. Colbert
S. A. Bowhill

August 1, 1981

Supported by
National Aeronautics
and Space Administration
Grant NSG 7506

Aeronomy Laboratory
Department of Electrical Engineering
University of Illinois
Urbana, Illinois

ABSTRACT

This work explores the feasibility of rewriting standard uniprocessor programs into code which contains no context-dependent branches. That is, this type of code (context independent code) would contain no branches that might require different processing elements to branch different ways.

In order to investigate the possibilities and restrictions of CIC, several programs were recoded into CIC and a four-element array processor was built. This processor (the Super-65) consisted of three 6502 microprocessors and the Apple II microcomputer. The results obtained were somewhat dependent upon the specific architecture of the Super-65 but within bounds, the throughput of the array processor was found to increase linearly with the number of processing elements (PEs). The slope of throughput versus PEs is highly dependent on the program and varied from 0.33 to 1.00 for the sample programs.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The needs of most computer users are constantly changing. These needs tend to demand faster and more powerful computers as the user puts the computer to more extensive use. Faster computers may be obtained either by improving the raw speed of the circuits and components or by using the same circuits in a more efficient architecture. Unlimited improvements in circuit speed cannot be expected due to fundamental physical constants, the most notable of these being the speed of light. Therefore, new approaches to computer organization must be found if projected demands of computer users are to be met, particularly in the area of large scientific problems.

In recent years, much attention has been given to unconventional organizations and various super-computers utilizing new concepts have been built [*Slotnick*, 1967]. An endless amount of questions and discussions is possible when the capabilities and handicaps of different organizations are compared. One can often find a specific application for which a given architecture excels as well as instances in which the same approach is ineffective. It is not the purpose of this work to make exhaustive comparisons of the capabilities and handicaps of different architectures. Only one particular organization will be dealt with: the array processor.

The array processor has been widely accepted by the computer community as a cost-effective approach in a particular but rather important set of applications [*Thurber and Wald*, 1975]. In this form of processor, high throughput is achieved by introducing parallelism, that is to say, several processors performing nearly identical operations. In this work, the array architecture is examined and a new approach to the design of an array processor is proposed in order to take advantage of the recent advent of

low-cost, high-performance microprocessors.

1.1 *What is an Array Processor?*

Illiac IV will be taken here as the conventional array processor. This section is not meant to be a complete description of Illiac IV and some familiarity with the work of *Barnes et al.* [1968] and of *Kuck* [1968] is assumed. Only a few basic concepts are considered here in order to set the stage for the discussion that follows.

Figure 1.1 shows the functional diagram of a single-processor computer. It consists of: (1) a memory to hold operands and instructons, (2) a control unit that fetches instructions from the memory, decodes them and issues control signals to (3) an arithmetic unit that performs the operations on operands taken from the memory. The most radical approach to parallelism would obviously be to replicate the elements shown in Figure 1.1 a number (n) times providing adequate interconnections between the elements. This is the multiprocessor approach [*Flynn*, 1972]. Although powerful, this organization leads to several implementation problems and as yet appears impractical for large n.

The expense of a multiprocessor architecture is primarily a result of the cost of the interface connecting each of the processors to each of the memories and the economic burden caused by the multiplicity of control units. This burden can be substantial as in a sophisticated classical machine; the control unit typically accounts for more than half of the total gate count [*Machado*, 1972].

These considerations lead one to the conventional array computer approach whose functional diagram is shown in Figure 1.2. Only the arithmetic units and memories are replicated and one single control unit (CU) drives the array of arithmetic units. Thus, an array processor is

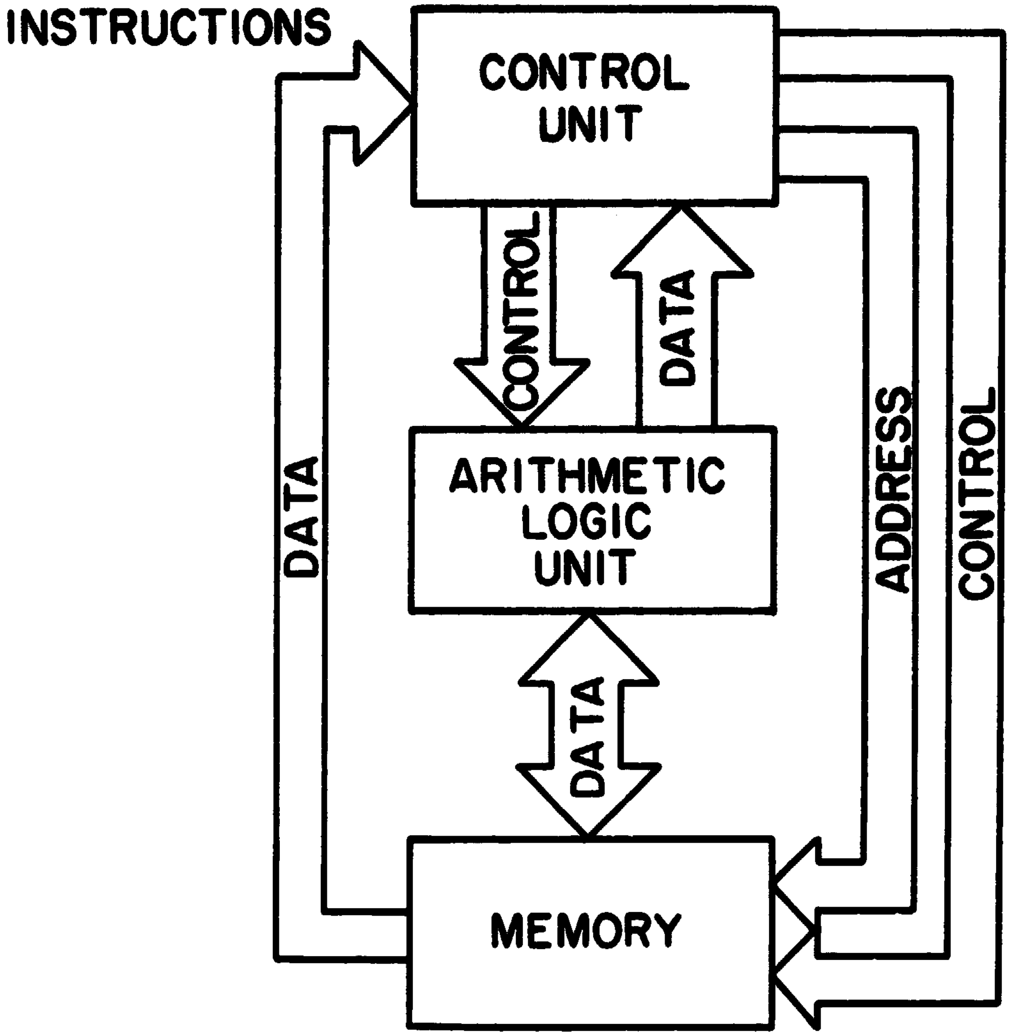

Figure 1.1 Single-processor computer.

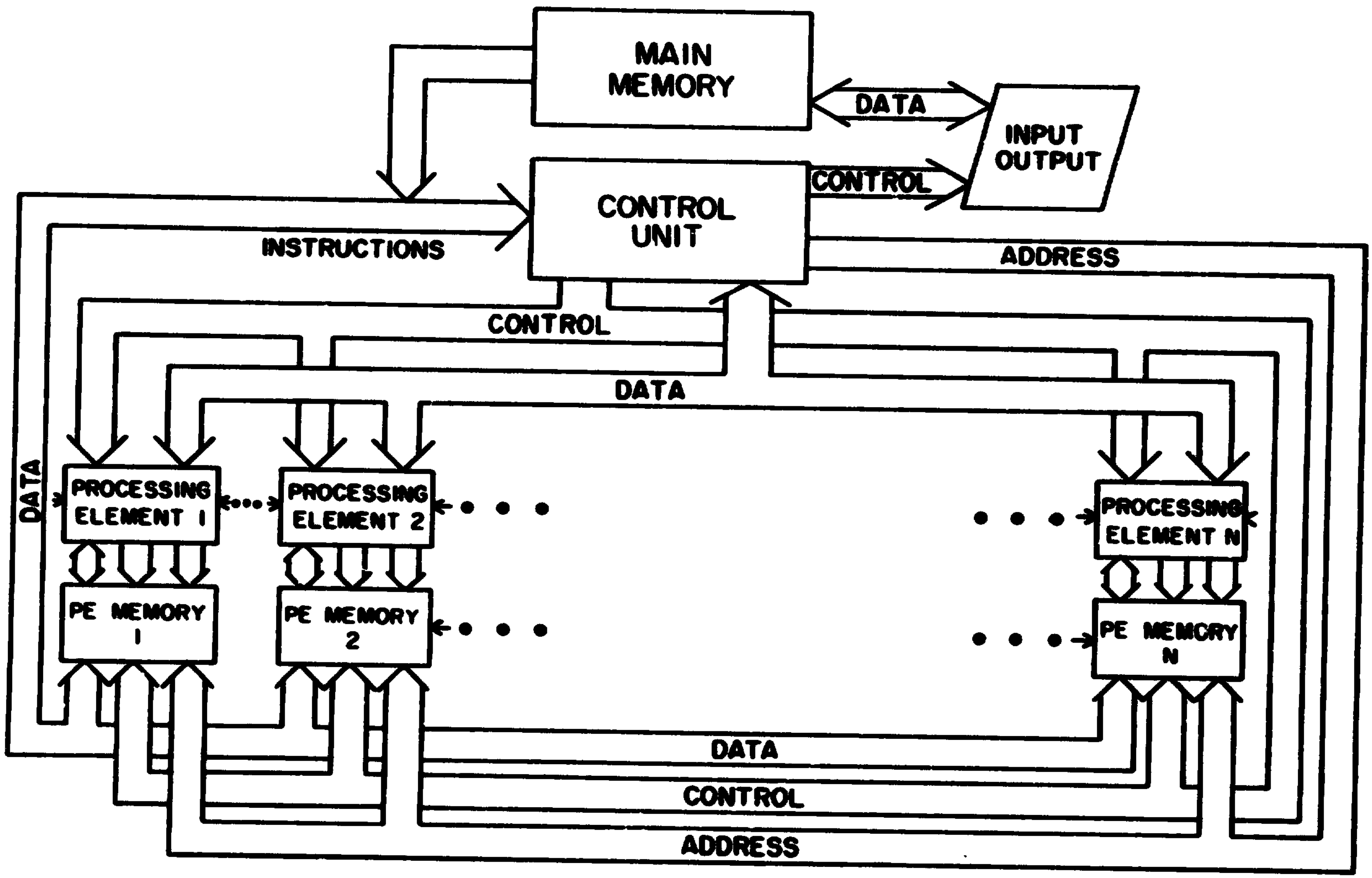

Figure 1.2 Conventional array computer.

characterized by the fact that a single instruction stream is executed simultaneously by any or all of the arithmetic units. For certain arithmetic units, the operation may have to be modified or suspended based on the contents of a gate or mask register in each processor. For this reason, the entire control unit cannot be made central as certain control decisions are operand-dependent. Therefore, a minimum amount of control is kept local and each arithmetic unit plus its local control is called a processing element (PE). The term processing unit (PU) is used to designate a PE with its processing element memory (PEM). Instructions can be stored either across the PEMs or in a special instruction memory.

One interpretation of the array-processor concept is that every PE performs precisely the same instruction on the same addresses in its own PEM. This constraint can be relaxed somewhat with the introduction of extra hardware to allow local indexing, mode control and routing. These concepts as they are commonly applied to an array processor (such as Illiac IV) will now be briefly described.

Local Indexing. The CU broadcasts an address to each PE. This address may be modified by each PE. In Illiac IV, for instance, an index register and address adder are provided with each PE. A central index register is also provided in the CU. The final operand address a_i for PE is the sum of the base address specified in the instruction, the contents of the central index register in the CU, and the contents of the local index register of the PE .

Mode Control. Although the goal of the array-processor structure is to be able to control the processing of a number of data streams with a single instruction stream, it is sometimes necessary to exclude some data streams or to process them differently. This is accomplished by allowing

each instruction to be locally modified by the PEs. The simplest form of mode control is to decide locally if central instruction I will be locally executed as I or as a no-op; i.e. each PE can be turned on or off. This is the only type of mode control available in Illiac IV. Complete mode-control capability would obviously result in a multiprocessor approach.

Routing. Routing is defined as the method by which PE_i may obtain an operand which is stored in PEM_j $(i \neq j)$. This need arises in many applications and therefore some way of routing operands from one PE to another is necessary. The simplest type of routing is to link PE_i to PE_{i-1} and PE_{i+1}. This is called neighbor routing. Non-neighbor routing is thus obtained by a sequence of neighbor routings. Illiac IV uses an advanced form of neighbor routing. This form is the four-nearest-neighbor routing. Each PE is able to communicate with the four PEs adjacent in the four directions (conventionally described as north, south, east and west).

1.2 *Motivation for Array Processors*

There are many reasons for using single-instruction-stream-multiple-data-stream (SIMD) architectures. *Thurber and Wald* [1975] contend that SIMD architecture is useful for large problems such as weather analysis and prediction, seismic data processing, phased-array-radar processing and picture processing. They also contend that problems with inherent data structure and parallelism such as solving systems of linear equations, Fourier transforms and systems of partial differential equations can be successfully executed on a SIMD machine. They further divide the advantages into the following categories.

1. Hardware

   a. Better use of hardware on highly parallel problems.

   b. Cost effectiveness due to the advent of LSI microprocessors.

c. Overcoming the speed of uniprocessors.

d. Reliability and graceful degradation of system.

2. Software

a. Simpler than for multiprocessor (MIMD).

b. Easier to construct large systems.

c. Less strict executive-function requirements.

These advantages are obtained at a price: such machines tend to be special-purpose, and any attempt to apply to inappropriate problems will likely be in vain.

1.3 *Issues and Objectives of This Study*

Assuming that the array processor is to be used in the proper environment, there still remain several issues to be resolved, such as processor-memory interconnection, interprocessor communication, software requirements (major modifications of uniprocessor program required, how context-dependent branches are handled), expandability of the system and fault tolerance of system.

1.3.1 *Processor-memory interconnection.* This study examines the limitations imposed by allowing communication from each of the processor elements to the shared memory (SM) only by way of a single address bus and a single data bus. This structure is attractive both economically and from the standpoint of system complexity. It is, in fact the simplest interconnection between several processors and a shared memory. As one can imagine, this simplicity imposes certain limitations. This study will seek to determine if the limitations imposed by this architecture are acceptable.

1.3.2 *Interprocessor communications.* As described earlier, most array processors have some form of interprocessor communication. A typical arrangement is to have a single bit channel of communication to each PE

immediately to the north, south, east and west. This research explores the problems which result when the only form of communication between two PEs is through the shared memory. This arrangement requires much less special hardware to be added to the basic processing unit but instead requires a WRITE to and a READ from the shared memory. On the other hand, the fact that only one processor can write to the shared memory at a time may adversely affect system throughput.

1.3.3 *Software requirements*. Because all processors share the same address bus in an array processor, each processor must execute the same instruction at the same time. If the programs to be executed were strictly linear with no branching, this would not be a restriction; however, most programs contain several branches and loops. Thus the uniprocessor program is not directly executable on the array processor. In Illiac IV [*Barnes et al.*, 1968; *Kuck*, 1968], as in most array machines, the programs reside in the shared memory and are specially compiled for the array by a host processor. The instruction stream seen by the individual PE is essentially uniprocessor code containing loops and branches. The Illiac IV allows local control to determine if the branch that the array is taking should be executed by that PE. If not, the PE executes no-ops until the array returns to execute the other branch. For code containing short context-dependent branches, the overall system throughput is not seriously degraded. However, if nested branches and long context-dependent branches are allowed, fewer and fewer PEs execute until all PEs are halted; the array then allows the waiting PEs to execute the next portion of code. *Flynn* [1972] has suggested that Minsky's Conjecture (that system throughput increases as $\log_2 n$ where n equals the number of PEs), may be accurate if, on the average, half of the remaining PEs continue to execute after a given

branch. As in the case of Illiac IV, the problems which the array processor is designed to solve do not force it into large nested branches often enough to produce such congestion.

One alternative, which removes the need to halt any PE, is to rewrite the original code so that it contains no context-dependent branches. That is, all context-dependent branches are recoded to allow the PE to execute the same instructions, but the data on which the instructions operate determine what operations are performed. This is not self-modifying code in the sense that the program resides in SM and is never altered. The present work investigates the feasibility of rewriting standard uniprocessor programs into code which contains no context-dependent branches, hereafter called Context Independent Code (CIC). This investigation also considers the limitations such recoding may have upon general usefulness of the system and on system throughput.

1.3.4 *Expandability of system.* Another aspect to be considered is that of expandability of the system. Many array systems are completely fixed as to the number of processors contained in the system. If a user desires to expand this system, the only solution is to add another complete system. An appealing aspect of the CIC software is that it allows the system to be expanded one processor at a time without requiring the system software to be completely rewritten. This is especially true if the number of processors is contaned as a variable within the program so that one simply increments the variable when a PE is added to the system. Also one is guaranteed that the system throughput increases linearly with n. This obviously contradicts Minsky's Conjecture, which advocates of array processing have been attempting to disprove for some time.

1.3.5 *Fault tolerance of system.* One final issue is that is fault-tolerance of the system. If one faulty processor causes the entire array to fail, the array will have a much higher failure rate than any one of the PEs. This means that if the system contains 1000 PEs, each with a failure rate of approximately .01%, the system has an unacceptable failure rate of 10%. However, if one is able to decouple the PEs to the extent that no one PE directly affects any other PE, then the failure rate is drastically reduced. More important than fault tolerance is fault detection. That is, one must be able to determine if the results the array is generating are valid. The concept of SIMD processing combined with CIC has the special feature that the address lines of all PEs must be the same at all times (other than when local indexing occurs); any departure by one PE is a certain indication of error.

The major objective of this thesis is to demonstrate that CIC recoding is feasible and attractive for some applications. In order to pursue this objective, a four-processor array computer was is built and utilized. Consequences of the parallel architecture are distinguished from those of the CIC recoding of the uniprocessor programs.

2. APPROACH TO A NEW ARRAY PROCESSOR

2.1 *Effect of Context-Dependent Branches on System Throughput*

Context-dependent branches reduce an array system's throughput significantly. With no context-dependent branches (assuming little or no memory contention), the throughput of *N* processors is *N* times that of the single-processor system. To illustrate why context-dependent branches reduce system throughput, consider an array of 32 processors. Allow this array to operate a program containing a single context-dependent branch with the length of the branch being 1/3 of the entire program (Figure 2.1). The entire array executes the first 1/3 of the program, then the array divides into two groups. One group desires to execute the left branch and the other group needs to execute the right branch. Obviously, the array can only execute one branch at a time and so one group of the array executes its branch while the other group is either disabled or performs no-ops. Then the groups reverse roles while the other branch is executed. Finally, the entire array executes the last 1/3 of the program. The time required for the 32 processors to execute this program is thus 4/3 the time required for a single PE. Hence, the throughput of the system for this program is 3/4 the throughput of the array when no context-dependent branches are encountered. However, the throughput achieved by this 32-PE array is 24 times the throughput of the single-processor system. One realizes from this example that long context-dependent branches will reduce the array performance much more than short context-dependent branches. One should note that a program containing several short context-dependent branches is usually preferred over a program containing fewer branches but with each branch having a significant length. An example might be a program with 20 context-dependent branches with each branch constituting .5% of the entire

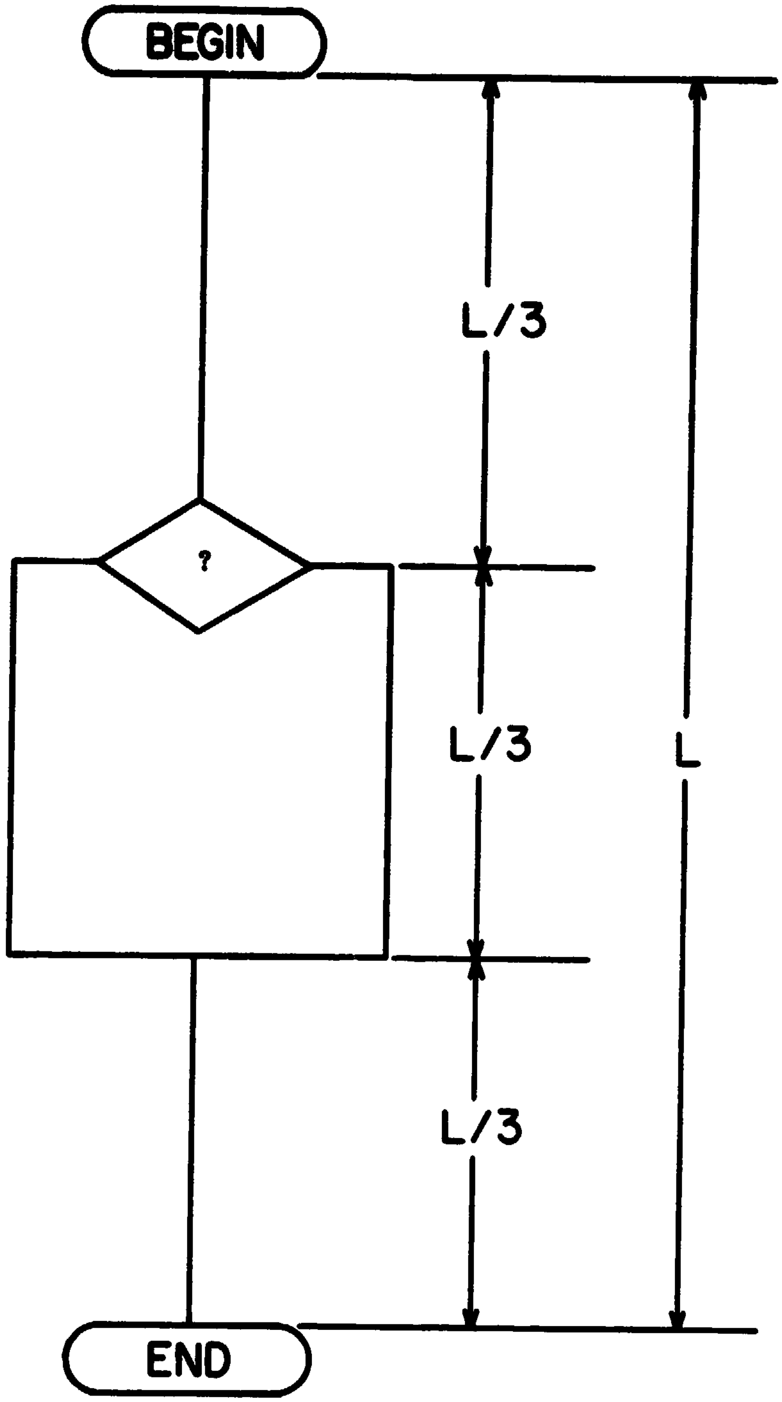

Figure 2.1 Program containing a single context-dependent branch of length L/3.

program (Figure 2.2). The time required for an array machine to execute this program would be 1.1 times the time required for a single PE. Hence, the array processor would be more than 90% efficient on this program and an array of 32 PEs would be able to process 32(1/1.1) or 29 times the amount of data in the same time as a single PE.

One final example is the program which contains nested context-dependent branches. Consider a program which contains N-level nested context-dependent branching. The array proceeds down one side of each decision until it reaches the innermost decision. It executes first one and then the other branch of the innermost decision until it has executed every possible branch of the tree. By comparison, the uniprocessor proceeds down the appropriate side of each decision, executing only those branches that are necessary.

Let us define S_{NA} to be the number of branches executed by an array processor for a program containing N-level nested context-dependent branching. One can see that for $N = 1$, $S_{1A} = 4$, for $N = 2$, $S_{2A} = 10$, and for $N = 3$, $S_{3A} = 22$ (Figure 2.3). That is, S_{NA} is the total number of branches contained in a program with N-level nested context-dependent branching. If one examines the flow diagrams carefully, one notes that S_{NA} exhibits the recursion

$$S_{NA} = 2 + 2S_{(N-1)A}$$

It is now asserted that:

$$S_{NA} = 2^{N+1} + 2^{N} - 2$$

Clearly, $S_{1A} = 2^2 + 2^1 - 2 = 4$, so we have a basis for induction.

Substituting $S_{(N-1)A} = 2^N + 2^{N-1} - 2$ into the recursion formula for S_{NA}

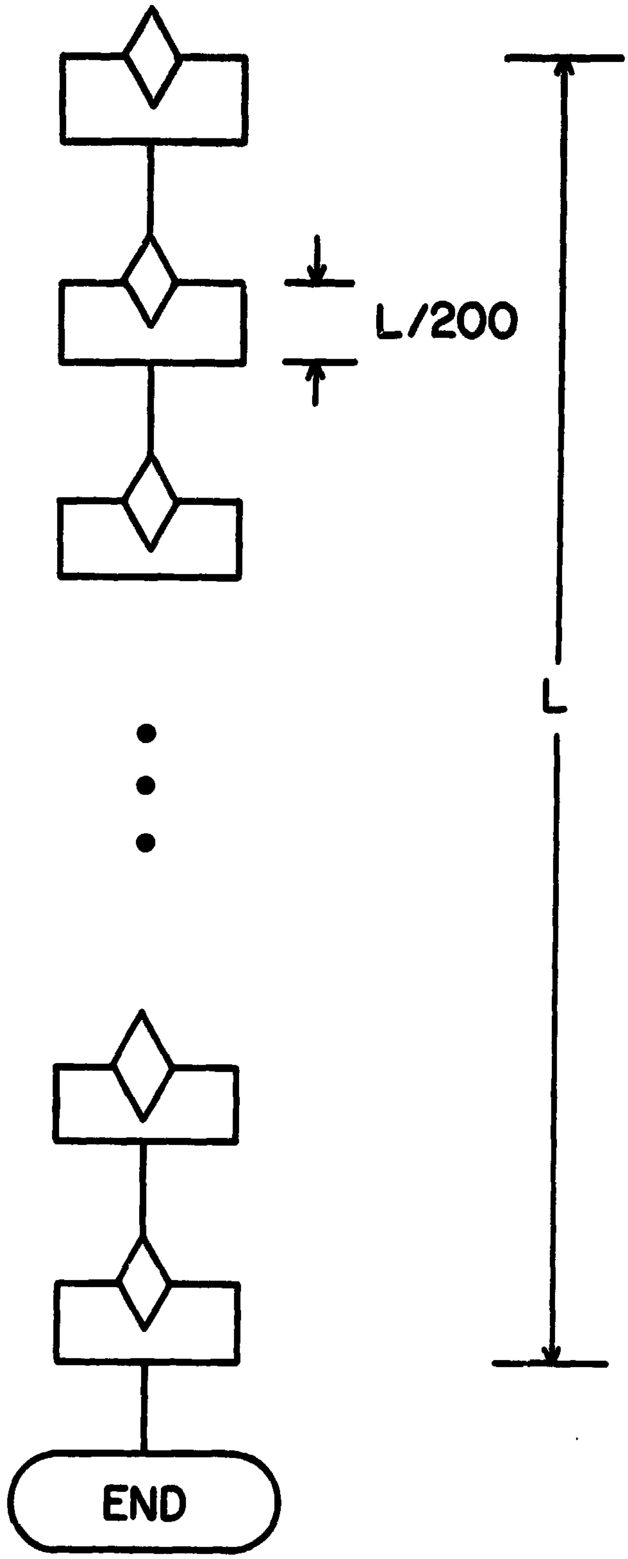

Figure 2.2 Program containing 20 context-dependent branches, each of length L/200.

yields the result

$$S_{NA} = 2 + 2(2^{N} + 2^{N-1} - 2)$$

or $$S_{NA} = 2 + 2^{N+1} + 2^{N} - 4$$

or $$S_{NA} = 2^{N+1} + 2^{N} - 2$$

Therefore, by induction, the assertion for S_{NA} has been proven.

Let us now define S_{NU} to be the branches executed by a uniprocessor for a program containing N-level nested context-dependent branching. From Figure 2.3, one notes that S_{NU} follows the recursion

$$S_{NU} = 2 + S_{(N-1)U}$$

It is now asserted that

$$S_{NU} = 2N + 1$$

and $$S_{(N-1)U} = 2(N-1) + 1$$

Substituting $S_{(N-1)U}$ into the recursion formula for S_{NU}, one obtains the result:

$$S_{NU} = 2 + [2(N-1) + 1]$$

$$S_{NU} = 2 + 2N-2 + 1$$

$$S_{NU} = 2N + 1$$

Therefore by induction, the assertion for S_{NU} has been proven.

If every branch of the program is assumed to be of equal length, the array processor will take S_{NA}/S_{NU} as long as the uniprocessor; this is because the array must execute all S_{NA} branches of the program instead of just S_{NU} branches as the single PE would. It is obvious that nested context-dependent branches can drastically reduce system throughput as

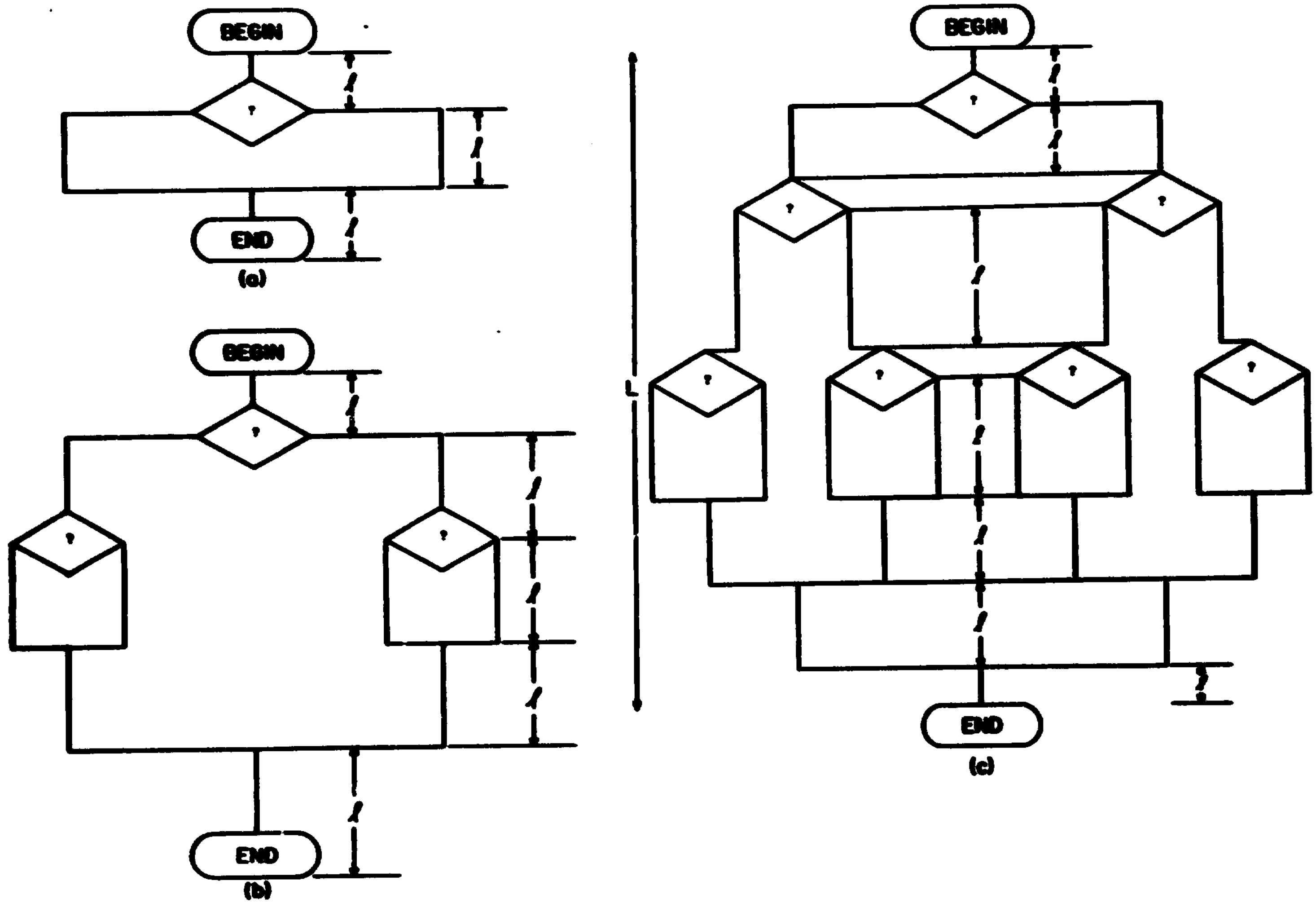

Figure 2.3 Program containing 3 level nested context-dependent branching

becomes large. The number of PEs required to allow the array processor to obtain the same throughput as the single processor for a program containing levels of nested context-dependent branching is:

$$n = \frac{2^{N+1} + 2^{N} - 2}{2N + 1}$$

Of course, this relationship assumes all branches to be of equal length and in a sense may be considered a worst case. However, one should still note that even for a reasonable level of nesting, an example of $N = 5$ the required number of PEs is greater than 8 (Figure 2.4). Hence, one should avoid nested context-dependent branches if at all possible.

2.2 *Software Consideration: Independent and Dependent Data Handling*

There are two principal methods of employing an array processor. The first method is to assume each processing element (PE) has its own source of data. That is, each processing unit is processing data which are independent of any other PE's data. This, in a sense is parallelism of the highest degree and is usually the simplest to implement as there need be little or no interprocessor communication. There are many applications for such array configurations.

The second method of using the array processor is to employ each PE on a subset of a larger problem. That is, the data given to each PE are related in some manner to the data given to the other PEs. An example might be that each PE is given a row of a large matrix and is given the job of multiplying that row of elements with each column of another matrix. In this way matrix multiplication may be performed rapidly.

Due to the complexity of array processors, array processor software is typically very difficult to read. Many times, the software will contain a substantial amount of special purpose instructions that are very machine-

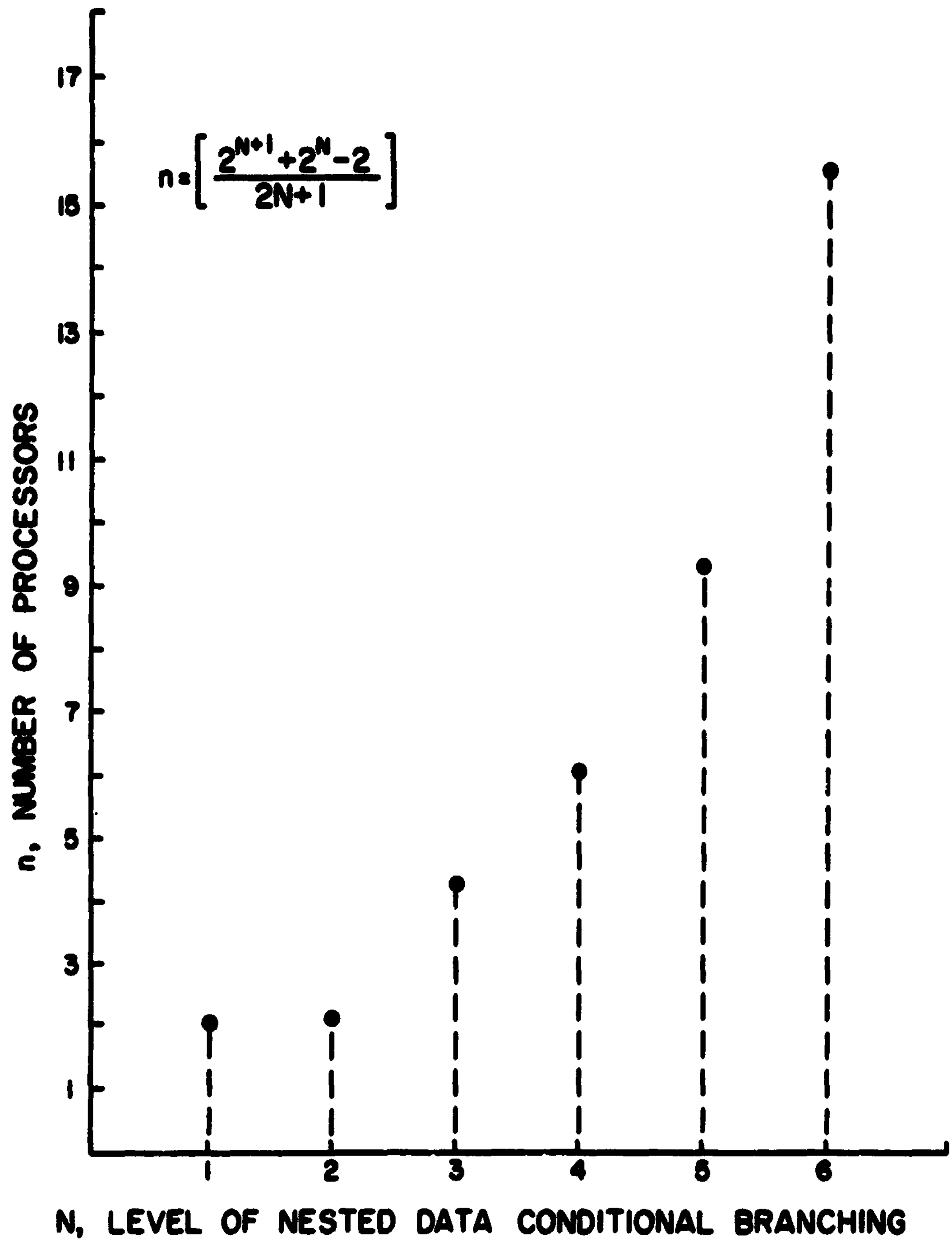

Figure 2.4 Graph of array throughput versus level of nested context-dependent branching.

dependent. In fact, designers of array machines have often decided to have the array processor execute only its own language. This language is usually optimized to a high degree for the particular array architecture. This obviously allows faster execution of programs written in and designed for that language. However, it forces any program written in another language to undergo a translation. These translations are seldom optimized and hence such translated programs are usually considerably less efficient.

Hence, if approaching a new array processor, one should carefully consider the compatibility of the new design with conventional languages. Also one should reduce the number of special instructions and other eccentricities to a minimum. This will serve to make the software more understandable and if one were to decide to use a different microprocessor, the conversion would be a much simpler task.

An array processor should ideally be designed for either separate independent data paths to each PE or a collection of dependent data. The principal difference between these two situations is that independent data paths typically require much less sophisticated interprocessor communications. Thus, if one knows that a great majority of the applications of this array processor will have independent data paths, the interprocessor communication channels may be simplified or eliminated.

2.3 *Context Independent Code and Its Implementation*

A new concept in the generation of array processors will be introduced here. This is the concept of Context Independent Code (CIC). The principle behind CIC is the elimination of context-dependent branches. One should note that CIC eliminates all context-dependent branches, not all conditional branches from the program. One cannot remove all conditional branches, since they allow a programmer to execute different segments of

code depending upon different conditions being present. Context-dependent branches are those conditional branches for which the data or condition may be different for different PEs. In particular, conditional branches which are used to cause the program to loop a certain number of times are not context-dependent, as all PEs will branch the same way every time. That is, the condition which the branch is based upon will be the same for all PEs. The conditional branches for which the condition will be different in different PEs must be recoded so that the program appears not to branch at all. For instance, the usual algorithm for multiplication shifts the multiplicand and tests each bit of the multiplier. If the multiplier bit is 1, the multiplicand is added to the partial product; if the bit is 0, the program skips the add instruction. For the conventional array processor, all PEs whose bit was 1 would execute the add instruction while the rest of the PEs were turned off.

This program written in CIC would cause all of the PEs to execute the add instruction, the difference being that those PEs whose bit was 1 would add the multiplier to the partial product while those PEs whose bit was 0 would add zero to the partial product.

This can be achieved in various ways, but of course one wishes to use the most efficient means possible. The most efficient method appears to be that of shifting each bit into the carry/borrow position and then subtracting that bit from the accumulator which has previously been set to zero. This results in either FF or 00 depending on whether the bit was 1 or 0. If one then ANDS the multiplicand with the previous result the outcome is either the multiplicand or zero. Thus, if one always adds the result of the iteration just described to the partial product, one will be adding the

multiplicand (if the bit was 1), or zero (if the bit was 0) to the partial product.

One might argue that forcing a PE whose multiplier bit is 0 to add the quantity 0 is no better than having it perform no-ops or turn itself off. From the standpoint of performing worthwhile tasks, this is true. However, this approach accomplishes two things which the previous approaches have not done.

1. All the PEs are constantly synchronized in a lock step mode. This does not require the programmer to sense which PEs are active and which are not. The procedure for determining the status of all the PEs can be somewhat time-consuming and may require considerable hardware.
2. The software is tailored to suit the array processor rather than tailoring an array processor to suit sequential software. This is more of a philosophical question at this time. Software that is designed to run on parallel array machines should prove to be more efficient than conventional uniprocessor software. However, it will probably take some time for programmers to adjust to array software and such software may be initially resisted. The need for array processors should overcome this initial resistance.

The implementation of Context Independent Code is almost totally free of restrictions. The single restraint is that all PEs must execute the same instructions at the same time. This means that the operands of the instruction determine which branch the PE is actually executing. This restriction eliminates the possibility of different PEs executing completely different branches at the same time with all the PEs doing worthwhile tasks all the time.

Implementation of CIC is very straightforward in that no special techniques are required. The programmer of the array is totally responsible for making certain the programs are successively written in Context Independent Code. At this time, the architecture does not have the capability to detect non-CIC programs and will attempt to execute any program that it is given. A more sophisticated architecture migat have a compiler that would flag non-CIC programs, but this is beyond the scope of this research. Examples of CIC programs will be included in a later chapter along with an explanation of the method one might use in recoding different programs into Context Independent Code.

2.4 *Input/Output Concepts*

Most array processors contain a host processor which controls input/output. Typically the host processor receives the input data and distributes it among the PEs. The host processor then requests the array to act on the input data and when the array has finished execution, the host processor then gathers the output data. The output data is then sent to a peripheral such as a tape, disc or video screen.

By using complete microprocessors as the PEs, one can allow the input data to come directly to each PE via a private input port. Also, if each PE generates sufficient output data, one may have each PE write to its own output port. That is, one might allow all PEs to receive their inputs simultaneously, perform the desired functions on the input and output the individual results, all in parallel. This, of course, is the best use of hardware and provides the highest possible throughput for the array.

If the input data are such that the outputs generated will be quite modest in number, a separate peripheral is not dedicated to each PE, but

instead the outputs are sent to shared memory where the controlling PE outputs them to a single peripheral.

The case may be that one cannot afford a separate peripheral for each PE, but one desires a greater throughput than an array with a single peripheral can provide. In this case, one may consider some special hardware that allows each PE to output to its own port. This special hardware can gather the outputs from several of the PEs to be stored in a given peripheral. As an example, assume that the system contains sixteen PEs with only four disc systems. The hardware transfers the outputs from four of the PEs to each of the disc systems. Various input/output arrangements are possible and the system designer can select the one most suitable for the type of application for which the array is intended.

2.5 *General Hardware Considerations*

In designing a new array processor, one must consider what technological advances are available. Of course, this is true in the case of classical computer design as well. However, an array processor has such a multiplicity of components that the opportunity for:

1. improved overall speed
2. reduced component cost
3. reduced power consumption
4. reduced chip count

is much greater than for a single processor computer.

Until recently the design of an array processor was restricted to very simple PEs [*Machado*, 1972] which typically had no local control except the ability to decide whether or not to execute a given instruction. All other controls resided in a single control unit (CU).

With the advent of inexpensive, single-chip microprocessors, one can

consider an array processor consisting of several microprocessors. Each microprocessor represents a single processing element (PE). Each microprocessor has a small private memory or processing element memory (PEM).

The PEs all share a large memory called shared memory (SM). The instruction stream comes from SM. As each of the microprocessors has all of the control logic necessary to operate as a separate computer, it is redundant to build a separate control unit (CU). Hence, one may designate one of the microprocessors as the controlling processor (CP) and eliminate the control unit. This approach also allows the possible implementation of a certain degree of fault tolerance since any of the PEs can become the CP if the original CP fails.

The decision to use microprocessors as the PEs restricts the word size of each PE to the word size of the microprocessor. However, most microprocessors allow for multiprecision arithmetic which allows one to achieve the degree of precision required for a given application. Of course, once one has decided to use microprocessors as PEs, a specific microprocessor must be selected. A discussion of how one may select the microprocessor is presented in a later section.

The next decision in the design of an array processor is the form of address and data bus system to be employed. One has the choice of a single bus system with both addresses and data multiplexed on the same bus or a two-bus system with separate data and address buses. The latter seems to be the most popular with microprocessor designers, basically because it simplifies the overall system.

Hence, one should recognize that a two bus system is simpler to use and easier to build into an array system. Next, one must consider the PE to shared memory (SM) connection. There are essentially two types of

connections:

1. Shared Address/Shared Data Bus

2. Multiple Address/Multiple Data Bus

The first type is the easiest to implement but imposes a possible bottleneck when the array contains more and more PEs. One should note that for the array processor, since every PE executes the same instruction, a READ from SM can be executed simultaneously. It is only the write to SM that must be executed sequentially. This is because one can only write one value to a specific address at a given time. Extra hardware might be used to place each PE's data word into a queue when a WRITE to SM is performed. The large number of WRITES to SM could then be executed in parallel with the PEs executing instructions which require access only to private memory. This would improve the effective transfer rate from the array to SM considerably. One requirement for this arrangement is that the values written to SM not be needed by any PE for a certain minimum time after the WRITE to SM was performed.

The second type of connection is considerably more difficult to implement completely. This type of connection requires SM to have the capability of communicating with several pairs of address and data bases. Memories of this type are commonly called true multiport memories. Multiport optical memories are now being researched [*Barnwell et al.*, 1978], but a true multiport as yet has not been placed on the market. One can simulate a memory with several ports by using extremely fast memory which is eight to ten times faster than standard memory chips. However, this does not solve the problem, when the number of PEs becomes greater than eight or ten.

Because the technology for multiport memory is not available at this time, one is forced to consider a shared bus system with perhaps some type

of queue to make the WRITE to SM appear to the individual PE to take about as long as a WRITE to its PEM. Certainly one would not want the WRITE to SM to take longer and longer as the number of PEs grows. However, the present work does not address the queue problem. Thus, in order to allow different PEs to write to SM, the architecture adopted requires each PE to become the CP in order to write to SM. This architecture has the unfortunate property that a WRITE to SM takes longer and longer as the number of PEs grows. This would not be significant if the PEs wrote only to SM to transfer final results at the end of every program.

One must consider what type of interprocessor communication is desired for the array processor. As previously noted, most array processors have nearest-neighbor connections. The most common form of communication is that of a single word. However, in order to provide a reasonable limit to this thesis, the design implemented allows no interprocessor communication other than through SM.

One final consideration is what method to use in transferring control from one PE to another. One can write the number of the PE desired to be CP to a specific address in memory. Alternatively one can use an unimplemented opcode of the microprocessor as a special instruction whose operand designates which PE is to become the CP. Finally, one can address a particular memory location (a so-called 'soft switch') in order to cause a particular PE to become the CP. The memory address is decoded to determine which PE is to be the CP. The first method uses a single memory location, but requires substantial hardware to latch the data word and decode which PE is desired. The second method uses no memory location but requires an unimplemented opcode. This could lead to difficulties if the chosen opcode were to be used in a later edition of the microprocessor. Also, most

microprocessor manufacturers will not guarantee what a microprocessor will do when it attempts to execute an unimplemented opcode. The third method, that of addressing a particular location in order to determine which PE is to be the CP is selected for use in the array processor design. The details of the array processor architecture are fully described in a later section.

2.6 *Selecting the Microprocessor for a Multi-Microprocessor System*

Selecting the microprocessor for a multi-microprocessor system involves many of the same considerations necessary when one wishes to design a single-microprocessor system.

One of the main considerations is whether or not a given microprocessor will be readily available, either for expanding the array or in case of component failure. *Sawin* [1977] provides a relatively complete list of available microprocessors.

Another important factor is compatibility. That is, whether or not the microprocessor is designed to be easily interfaced both with peripheral chips and with other microprocessors of the same type. With the variety of microprocessors available today, an extensive comparison of all of the possible choices would be quite lengthy. However, one can reduce the selection considerably if one is interested only in general-purpose microprocessors that are reasonably inexpensive. This removes special-purpose microprocessors from consideration. Also, as yet, 16-bit microprocessors are relatively expensive and are not used extensively enough for them to be readily available. Thus, one should not attempt to use 16-bit microprocessors in a multi-microprocessor environment until they are more readily available and their unit price is reduced somewhat as will inevitably occur. One might consider a possible modificaton of the architecture at a

later time to allow use of 16-bit microprocessors rather than the standard 8-bit microprocessors which are in widespread use today.

Now one would like to reduce the overall chip count for the entire array. Single-chip 8-bit microprocessors are now readily available and it seems appropriate to select a single-chip processor if at all possible.

Another factor to be considered in narrowing the selection of a microprocessor is that of versatility. Does the processor allow straightforward implementation of multi-byte arithmetic? This capability is extremely important in an array processor since many applications of array processors require processing 16- and 32-bit data. Similarly, one should consider the architecture of the proposed microprocessor. How many registers are available to the programmer? How many different addressing modes does the processor support? Does the processor have the capability of implementing a stack? What address range is the microprocessor capable of addressing? What type of interrupt capability does the microprocessor have? Is the programmer able to halt the microprocessor? What are the consequences of halting the microprocessor? Does the microprocessor allow for direct memory access (DMA) by other devices?

One important area that should be analyzed carefully is the instruction set of the microprocessor. Does the instruction set contain all the essential instructions required to perform arithmetic, logical and program control functions? How efficient is the PE with respect to the number of machine cycles required per instruction? Does the microprocessor require several machine cycles in order to execute even the simplest instructions? If this is so, the microprocessor may be actually much slower than another microprocessor that has a slower clock rate but requires fewer machine cycles per instruction.

One should also consider what development aids are available for a given microprocessor. For instance, are there entire systems based on a given microprocessor such that one could use an assembler, editor, and other diagnostic aids? Also, the ability to use a commercial system with keyboard, video and monitor already in working order is invaluable.

In summary then, the optimum microprocessor for a multi-microprocessor system would:

1. be a single-chip, 8-bit microprocessor
2. be readily available
3. be easily interfaced with various peripherals as well as other microprocessors of the same family
4. allow multiple precision arithmetic
5. have as many registers as possible
6. have many modes of addressing
7. be capable of addressing as large an address range as possible
8. support sufficient interrupt levels
9. have the ability to implement a stack and thus allow subroutines to be used effectively
10. normally require few machine cycles in order to execute a given instruction
11. contain a relatively powerful instruction set that allows the PE to be very versatile
12. operate at a clock rate that is competitive with other possible choices.

3. HARDWARE ASPECTS OF THE SUPER-65 MULTI-MICROPROCESSOR SYSTEM

3.1 *Attributes of the 6502 Microprocessor*

In the previous section, the best choice of the microprocessor was reduced to readily available, single-chip, 8-bit, general purpose microprocessors. It was also noted that all microprocessors considered should operate at a competitive clock rate and be capable of addressing a large address range. These restrictions reduce the selection of suitable microprocessors to essentially three. They are:

(1) Zilog Z-80

(2) Motorola MC6800

(3) MOS Technology 6502

Each of these microprocessors is a general purpose, single chip, 8-bit microprocessor. Each requires a single +5-volt supply and is TTL compatible. These microprocessors may operate at various clock rates and in order to compare them fairly, one must consider instruction execution times at the same clock rate. A standard clock rate is one MHz.

The minimum instruction execution time is one type of comparison which provides the system designer with a clearer understanding of the relative performance of different microprocessors. With a clock rate of 1 MHz, the Z-80 has a minimum instruction execution time of 4 microseconds [*Garland*, 1979]. The MC-6800, and the 6502 both have a minimum instruction execution time of 2 microseconds with a clock rate of 1 MHz [*Artwick*, 1980].

The Z-80 has 158 instruction opcodes, two 16-bit index registers, a 16-bit stack pointer and 14 general-purpose 8-bit registers [*Borden*, 1978]. The Z-80 has the following addressing modes:

(1) Implied Addressing

(2) Immediate Addressing

(6) Extended or Absolute Addressing

(7) Modified Page Zero Addressing

(3) Extended Immediate Addressing
(4) Register Addressing
(5) Register Indirect Addressing
(8) Relative addressing
(9) Indexed Addressing
(10) Bit Addressing

However, the Modified Page Zero Addressing mode is used only for one instruction, the Restart Page Zero instruction. Also the Bit Addressing mode is used solely to set, clear or test bits in a given word. Finally, the Extended Immediate Addressing mode simply indicates that the immediate operand is 16 bits rather than 8 bits. Thus, the Z-80 actually has only 7 different addressing modes. Another point to consider is that many of the Z-80 instructions are due to the large number of registers available and do not give one a greater variety of instructions as one might be tempted to think.

The MC 6800 has 72 instructions, one 16-bit index register, a 16-bit stack pointer and two 8-bit accumulators. The MC 6800 has the following addressing modes:

(1) Implied Addressing
(2) Immediate Addressing
(3) Zero Page Addressing
(4) Absolute Addressing
(5) Relative Addressing
(6) Zero Page Indexed Addressing

A major deficiency of the 6800 is its lack of an indirect addressing mode. One also notes that indexing can only be done in the zero-page mode.

At this point, a historical perspective helps in understanding the evolution of these three microprocessors. *Scanlon* [1980] relates that in 1973, Intel Corporation introduced a second generation 8-bit microprocessor called the 8080. The 8080 was designed with a calculator-like architecture with eight scratch-pad registers, an internal stack register and special input and output instructions.

Motorola Inc. saw the tremendous microprocessor market potential

evolving, and decided to make an entry of their own. They had two choices. They could challenge Intel Corporation on their ground by producing a new and improved version of the 8080, as Zilog Inc. did in 1976 with the Z-80. The other choice was to design a more advanced microprocessor. Realizing the difficulty of the first approach, Motorola decided to challenge Intel Corporation with a superior product.

For the 6800 microprocessor, Motorola abandoned the calculator-like register-oriented architecture of the 8080, and adopted a classic minicomputer-like memory-oriented architecture. As a result, the 6800 has fewer (and easier to understand) instructions, with more addressing options than the 8080.

The preceding brief overview is necessary in order to set the stage for introducing our chosen microprocessor, the 6502. The 6502 device was designed by ex-employees of Motorola who saw that advances in processes, coupled with a few architectural and software changes, could result in a potentially highly marketable 6800-like microprocessor. They joined a calculator-chip company called MOS Technology.

The MOS Technology design team had two objectives in mind for their next generation microprocessor--low cost and high performance. They reduced the complexity of the basic 6800 design as much as possible to increase chip yield. Design changes included eliminating one of the two accumulators in the 6800 and its tristate address output buffers. They replaced the 16-bit index register of the 6800 with two separate 8-bit index registers and discarded some of the lesser-used instructions of the 6800.

The elimination of these instructions opened up some instruction-decode space and permitted the designers to provide the 6502 microprocessor with 13 addressing modes, 7 more modes than the 6800. These modes give the

6502 capabilities that are normally found only in larger computers. This addressing capability is complemented by the extremely fast speeds at which the 6502 can execute instruction sequences. This speed is primarily due to the fact that the 6502 is designed with a pipelining technique in which the microprocessor fetches the next instruction before it is done processing the current instruction. Additionally, the design team added a decimal mode select instruction and control bit that allows the 6502 to operate on either binary or decimal data. This means that the programmer does not have to remember to write in a 'decimal adjust' instruction after every addition or subtraction. Also the newer depletion-load technology was employed, which gives the 6502 cleaner switching characteristics and lower power dissipation. The 6502 typically dissipates 250 mW versus 500 mW typical for the 6800. This technology also results in better noise immunity.

The addressing modes for the 6502 are:

(1) Implied Addressing

(2) Immediate Addressing

(3) Zero Page Addressing

(4) Zero Page Indexed (Xreg) Addressing

(5) Zero Page Indexed (Yreg) Addressing

(6) Absolute Addressing

(7) Absolute Indexed (Xreg) Addressing

(8) Absolute Indexed (Yreg) Addressing

(9) Relative Addressing

(10) Indirect Indexed (Yreg) Addressing

(11) Indexed Indirect (Xreg) Addressing

(12) Indirect Addressing

(13) Accumulator Addressing

The 6502 has 56 different types of instructions, with 151 different instruction opcodes (i.e., 151 different instructions). One readily sees that the 6502 obviously has the most powerful addressing capability of the three microprocessors. The 6502 instruction set is almost equal in size with that of the Z-80. One key advantage of the 6502 over the Z-80 is that for the same clock rate, the 6502 is at least twice as fast as the Z-80. As microprocessors are built to run faster and faster, the determining factor is not how fast an instruction is executed in absolute time, but how many machine cycles are required. In this respect, the pipelining done in the 6502 makes the 6502 instruction execution time with respect to machine cycles very efficient.

The Zero Page Addressing capability allows the 6502 to use all 256 locations in page zero of memory as though they were registers. This allows extreme versatility in programming the 6502. In fact, if the programmer uses this feature of the 6502 properly, it is possible to realize up to a nother factor of two increase in speed over the other microprocessors. The page zero of memory can be utilized by the 6502 as more powerful computers use cache memories. This feature alone makes the 6502 an excellent choice for an array processor. When one considers the minimal power requirements of the 6502, the superior addressing capability, and the higher throughput due to pipelining, there is absolutely no better choice.

Although the decision of the microprocessor has already been made, one should point out that the 6502 does indeed satisfy and in some cases, exceeds the requirements set forth in the preceding section. The 6502 is the most efficient of the three microprocessors both in terms of speed and power. It has the most powerful addressing capability of the three. It has no strictly general purpose internal registers but more than makes up

for this by the ablity to use the entire zero page of memory as registers and by having not one but two index registers. It has stack capabilities so that one may utilize both subroutines and interrupt routines. It has the standard two level interrupt capability. That is, it has both maskable and non-maskable interrupts. It has on-board clock circuitry which reduces the components necessary for a minimal microcomputer system [*Camp et al.*, 1979]. Its instruction set provides the programmer with all the essential resources for programming the most diverse programs. The 6502 is extremely easy to interface both with other 6502 microprocessors and with other 6500 series components such as input/output chips. The address bus of the 6502 always has a valid memory address. This allows for much easier synchronization of several microprocessors. The 6502 has two control lines called 'READY' and 'SYNC' which allow the possibility of single stepping through a program. If the READY line is pulled low during phase one of a SYNC high cycle, single-step operation of the 6502 can be achieved. The 6502 provides a vectored reset operation that allows one to program a unique initialization routine to fit ones own needs. The 6502 has had widespread use, not just as a microprocessor in user-designed systems but in many commercial microcomputers such as the Apple, AIM-65, SYM, KIM, PET and many others. Thus the 6502 is easily available and is compatible with many commercial microcomputers.

Obviously, the final decision of which microprocessor to use is somewhat subjective, but it is interesting to note that the latest 16-bit microprocessor from Zilog, Inc., the Z-8000, has a memory-oriented architecture (such as the 6502) which represents a solid break with the 8080/ Z-80 architectural design concept of register-oriented microprocessors. Also, this might indicate that a design with the 6502 microprocessor would

be more easily converted to the newer 16-bit microprocessors should one ever decide to modify the array processor.

3.2 *The Apple II Microcomputer System*

The Apple II microcomputer system is a versatile microcomputer that employs the 6502 as its microprocessor. The Apple II has the capability of the full 64K of memory, using dynamic RAM to reduce cost and power consumption. The Apple II provides the 6502 with a 1.023 MHz clock signal which is supplied to the phase zero (ϕ_0) input of the 6502. The microprocessor uses its address and data buses only when phase zero is high. When phase zero is low, the microprocessor is doing internal operations and does not need the data and address buses. The Apple II designers allow the memory to be refreshed at a 3.5 MHz rate, when phase zero (ϕ_0) is low. In this way, memory refresh is entirely transparent to the 6502. *Espinosa* [1979] explains the system timing entirely and provides a schematic of the Apple II that is invaluable to the system designer.

The 16-bit address bus lines are buffered by tristate buffers. The address lines are held open only during a DMA cycle and are active at all other times. A DMA cycle also halts the 6502. The address on the address bus becomes valid about 300 nsec after phase one (complement of phase zero) goes high and remains valid through all of phase zero (see Figure 3.1).

The 8-bit data bus lines are buffered by bi-directional tristate buffers. Data from the microprocessor is put onto the data but about 300 nsec after phase one (ϕ_1) and READ/$\overline{\text{WRITE}}$ both drop to zero. At all other times, the microprocessor is either listening to or ignoring the data bus.

The RDY, RES, IRQ and NMI lines to the microprocessor are all held high by 3.3k Ohm resistors to +5V (see Figure 3.2). These lines also

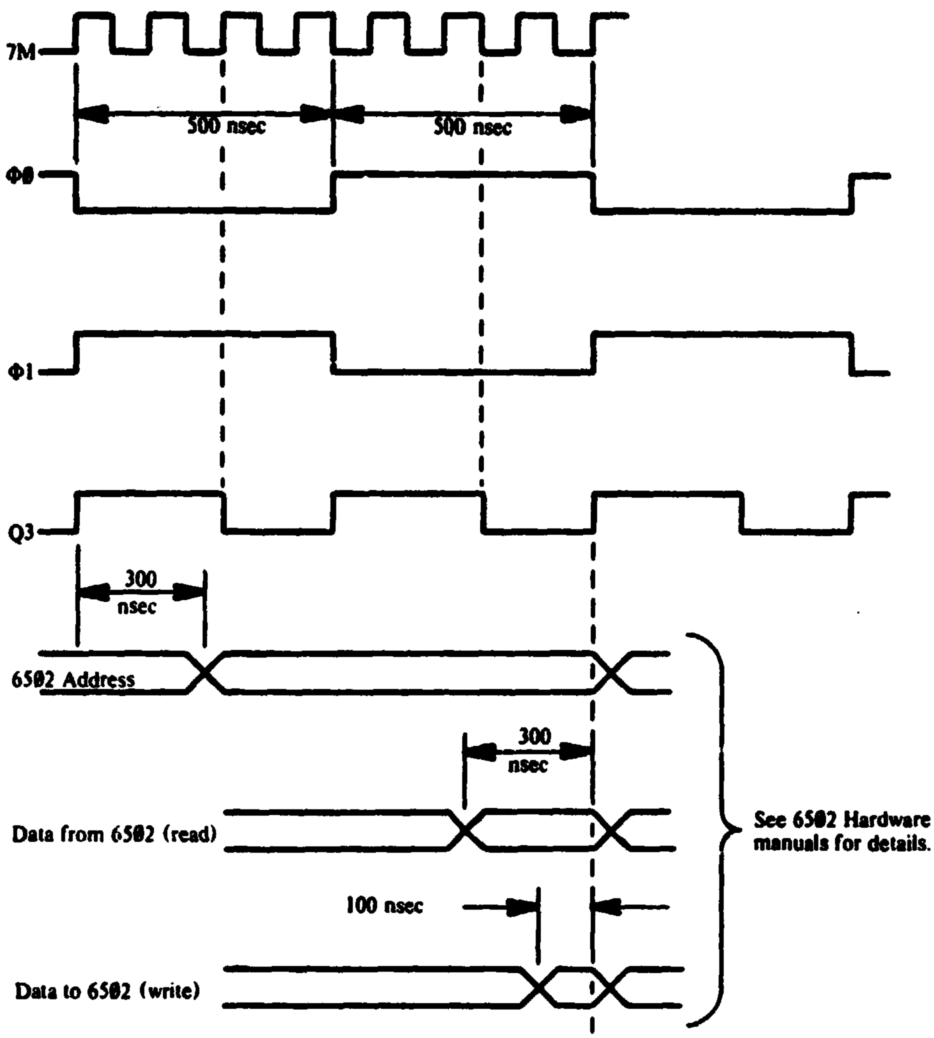

Figure 3.1 6502 timing signals.

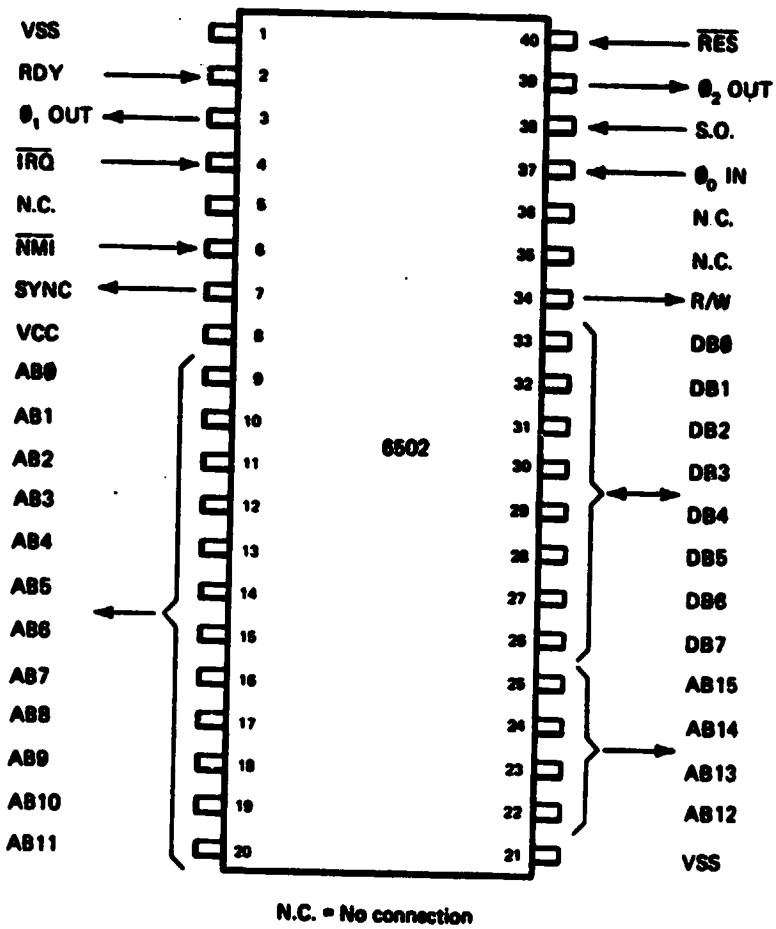

Figure 3.2 6502 microprocessor pinout designation (courtesy MOS Technology).

appear on the peripheral connectors (see Figure 3.3 and 3.4). The Set Overflow line to the microprocessor is permanently tied to ground.

All timing signals are derived from a 14.318 MHz master oscillator output. The 7.159 MHz intermediate timing signal and the 1.023 MHz signals phase zero and phase one are the only timing signals employed in the design of the Super-65 array processor.

The Apple can support up to six 2K x 8 mask programmed READ-Only memory chips. One of the six ROMS is enabled whenever the microprocessor's address bus holds an address between $D000 and $FFFF. Thus, the address range $D000-$FFFF is reserved for ROM.

The Apple supports up to 48K x 8 Random Access Memory (RAM) or READ/WRITE memory. As previously mentioned, this RAM is dynamic and is refreshed automatically during every phase one (ϕ_1) cycle. The Apple supports a sophisticated video system, but this need not be discussed in detail here.

The Apple provides two female miniature phone jacks that allow one to connect the Apple to a normal cassette tape recorder. In this way, one can store user programs permanently on tape without incurring the expense of a complete tape system.

The Apple provides users with eight peripheral connectors along the back of the Apple's main board. These slots are designed to allow the user more sophisticated resources such as disk drives, the ability to program in high level languages directly, etc. Also, the Apple designers give the user the option of plugging in proto-boards containing user-designed circuits.

The Apple designers intend slot zero as a special purpose slot so that many of the options available to the other seven slots are unavailable to

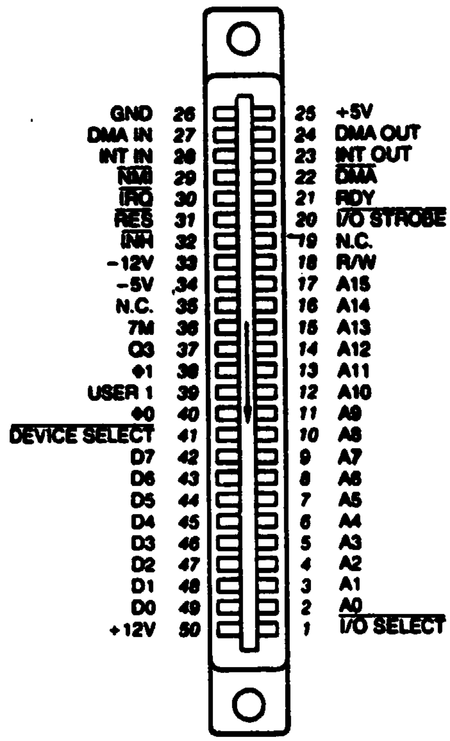

Figure 3.3 Peripheral connector pinout.

Pin:	Name:	Description:
1	$\overline{\text{I/O SELECT}}$	This line, normally high, will become low when the microprocessor references page \$C*n*, where *n* is the individual slot number. This signal becomes active during $\Phi 0$ and will drive 10 LSTTL loads*. This signal is not present on peripheral connector 0.
2-17	A0-A15	The buffered address bus. The address on these lines becomes valid during $\Phi 1$ and remains valid through $\Phi 0$. These lines will each drive 5 LSTTL loads*.
18	R/$\overline{\text{W}}$	Buffered Read/$\overline{\text{Write}}$ signal. This becomes valid at the same time the address bus does, and goes high during a read cycle and low during a write. This line can drive up to 2 LSTTL loads*.
19	SYNC	On peripheral connector 7 *only*, this pin is connected to the video timing generator's SYNC signal.
20	$\overline{\text{I/O STROBE}}$	This line goes low during $\Phi 0$ when the address bus contains an address between \$C800 and \$CFFF. This line will drive 4 LSTTL loads*.
21	RDY	The 6502's RDY input. Pulling this line low during $\Phi 1$ will halt the microprocessor, with the address bus holding the address of the current location being fetched.
22	$\overline{\text{DMA}}$	Pulling this line low disables the 6502's address bus and halts the microprocessor. This line is held high by a 3KΩ resistor to +5v.
23	INT OUT	Daisy-chained interrupt output to lower priority devices. This pin is usually connected to pin 28 (INT IN).
24	DMA OUT	Daisy-chained DMA output to lower priority devices. This pin is usually connected to pin 22 (DMA IN).
25	+5v	+5 volt power supply. 500mA current is available for *all* peripheral cards.
26	GND	System electrical ground.
27	DMA IN	Daisy-chained DMA input from higher priority devices. Usually connected to pin 24 (DMA OUT).
26	INT IN	Daisy-chained interrupt input from higher priority devices. Usually connected to pin 23 (INT OUT).
29	$\overline{\text{NMI}}$	Non-Maskable Interrupt. When this line is pulled low the Apple begins an interrupt cycle and jumps to the interrupt handling routine at location \$3FB.

Figure 3.4 Peripheral connector descriptions.

30	$\overline{\text{IRQ}}$	Interrupt ReQuest. When this line is pulled low the Apple begins an interrupt cycle only if the 65∅2's I (Interrupt disable) flag is not set. If so, the 65∅2 will jump to the interrupt handling subroutine whose address is stored in locations \$3FE and \$3FF.
31	$\overline{\text{RES}}$	When this line is pulled low the microprocessor begins a RESET cycle (see page 36).
32	$\overline{\text{INH}}$	When this line is pulled low, all ROMs on the Apple board are disabled. This line is held high by a 3KΩ resistor to +5v.
33	−12v	−12 volt power supply. Maxmum current is 200mA for all peripheral boards.
34	−5v	−5 volt power supply. Maximum current is 200mA for all peripheral boards.
35	COLOR REF	On peripheral connector 7 *only*, this pin is connected to the 3.5MHz COLOR REFerence signal of the video generator.
36	7M	7MHz clock. This line will drive 2 LSTTL loads*.
37	Q3	2MHz asymmetrical clock. This line will drive 2 LSTTL loads*.
38	Φ1	Microprocessor's phase one clock. This line will drive 2 LSTTL loads*.
39	USER 1	This line, when pulled low, disables *all* internal I/O address decoding**.
40	Φ∅	Microprocessor's phase zero clock. This line will drive 2 LSTTL loads*.
41	$\overline{\text{DEVICE SELECT}}$	This line becomes active (low) on each peripheral connector when the address bus is holding an address between \$C∅*n*∅ and \$C∅*n*F, where *n* is the slot number plus \$8. This line will drive 10 LSTTL loads*.
42-49	D∅-D7	Buffered bidirectional data bus. The data on this line becomes valid 300nS into Φ∅ on a write cycle, and should be stable no less than 100ns before the end of Φ∅ on a read cycle. Each data line can drive one LSTTL load.
50	+12v	+12 volt power supply. This can supply up to 250mA total for all peripheral cards.

Figure 3.4 (cont.) Peripheral connector descriptions.

slot zero. Each slot is given sixteen locations beginning at $C080 for general input and output purposes. For slot zero, these sixteen locations are $C080-$C08F; for slot one they are $C090-C09F, etc. Whenever the address on the address bus is in a given slot's allocated range, pin 41 (called Device Select) goes low. This alerts the particular card that the address is somewhere in that peripheral card's 16-byte allocation.

Each peripheral slot also has reserved for it one 256-byte page of memory. This page is usually used to house 256 bytes of ROM, which contains driving programs or subroutines for the peripheral card. The page of memory reserved for each peripheral slot has the page number $C*n*, where is the slot number. The signal on pin 1 (called I/O Select) of each peripheral slot becomes active (drop to ground) when the microprocessor is addressing an address within that slot's reserved page.

The 2K memory range from location $C800 to $CFFF is reserved for a 2K ROM or PROM on a peripheral card, to hold large programs, etc. The expansion ROM space also has the advantage of being absolutely located in the Apple's memory map, which gives one more freedom in writing interface programs. This PROM space is available to all peripheral slots and more than one card can have an expansion ROM. However, only one expansion ROM can be active at one time. The expansion ROM typically requires 2 enable inputs. A suggested method is to use pin 1 (I/O Select) as one enable and pin 20 (called I/O Strobe) as the other. The I/O Strobe line goes low (active) when the address bus contains an address within the expansion ROM space (i.e., between location $C800 and location $CFFF).

Thus, each peripheral card has available the buffered address bus, buffered data bus, buffered READ/$\overline{\text{WRITE}}$ line, the READY line, the Non-Maskable Interrupt line, the Interrupt Request (IRQ) line, and the Reset

line. Other leads available include: 1) the $\overline{\text{DMA}}$ line which disables the 6502's address bus and halts the microprocessor, 2) the $\overline{\text{I/O Select line}}$ which goes low (active) on the peripheral card when the address bus contains an address within page \$Cn, when n is the particular slot number, 3) the Device Select line which goes active (low) on peripheral connector when the address bus is holding an address between \$C0n0 and \$C0nF, where is the particular slot number plus eight and 4) the $\overline{\text{I/O Strobe}}$ line which becomes active (low) on all connectors, when the address on the address bus is between \$C800 and \$CFFF. Of course, all of the peripheral connectors have phase zero (ϕ_0), phase one (ϕ_1), and the 7 MHz clock signals available for synchronization of the cards with the 6502.

The Apple System monitor acts as a supervisor of the system. From the monitor one may look at one, some, or all memory locations; one can write programs in Machine and Assembly languages to be executed directly by the Apple's microprocessor; one crn save data and programs onto cassette tape or a floppy disk and read them back in again; one can move and compare several bytes of memory with a single command; and one can leave the monitor and enter any other program or language available on the Apple.

There is a program within the monitor which allows one to type programs into the Apple in assembly level language. This program is called the Apple Mini-Assembler. It is a 'mini'-assembler because is cannot understand symbolic labels, something that a full assembler must do. For details in using the Apple Mini-Assembler one should refer to the Apple II Reference Manual (pp. 49-51).

The Apple II Monitor provides facilities for stepping through programs both in single step and trace mode. Also, one is able to examine the contents of the 6502's internal registers after each instruction is executed.

This allows one to properly debug difficult programs in a very straightforward manner.

One can see from the above description, that the Apple II gives the user a very powerful microcomputer with all the necessary facilities to write, assemble, debug and execute programs varying from machine-level to high-level languages. It also provides more than adequate means for extensive use of peripherals. Thus, the Apple II is an excellent choice for implementing an array processor as it is not only versatile but economical as well.

3.3 *Architecture of the Overall System*

The previous two sections provided the necessary information for the two major building blocks of the Super-65 System; the 6502 microprocessor and the Apple II microcomputer. One can now proceed to the description of the Super-65 System.

The Super-65 Multi-Microprocessor System consists of four 6502 microprocessors, one being the Apple II 6502, with three others. Each 6502 is given a private random access memory of dimension 1K by 8 bits. This memory resides in the lowest portion of the 6502's memory map. The rest of the 6502's memory (the upper 63K) resides in the Apple II. This will be both RAM and the Apple II Monitor ROM. The combined RAM and ROM are jointly designated as Shared Memory (SM). One can see from the block diagram (Figure 3.5), that the architecture of the Super-65 is relatively simple. Each processor is able to access its private memory at any time but only one processor is able to access the Shared Memory Address Bus at a time. All processors may access the SM Data Bus on a READ. However, for obvious reasons, only one processor may access the SM Data Bus during a WRITE operation. One notes that the architecture allows for both shared input/output

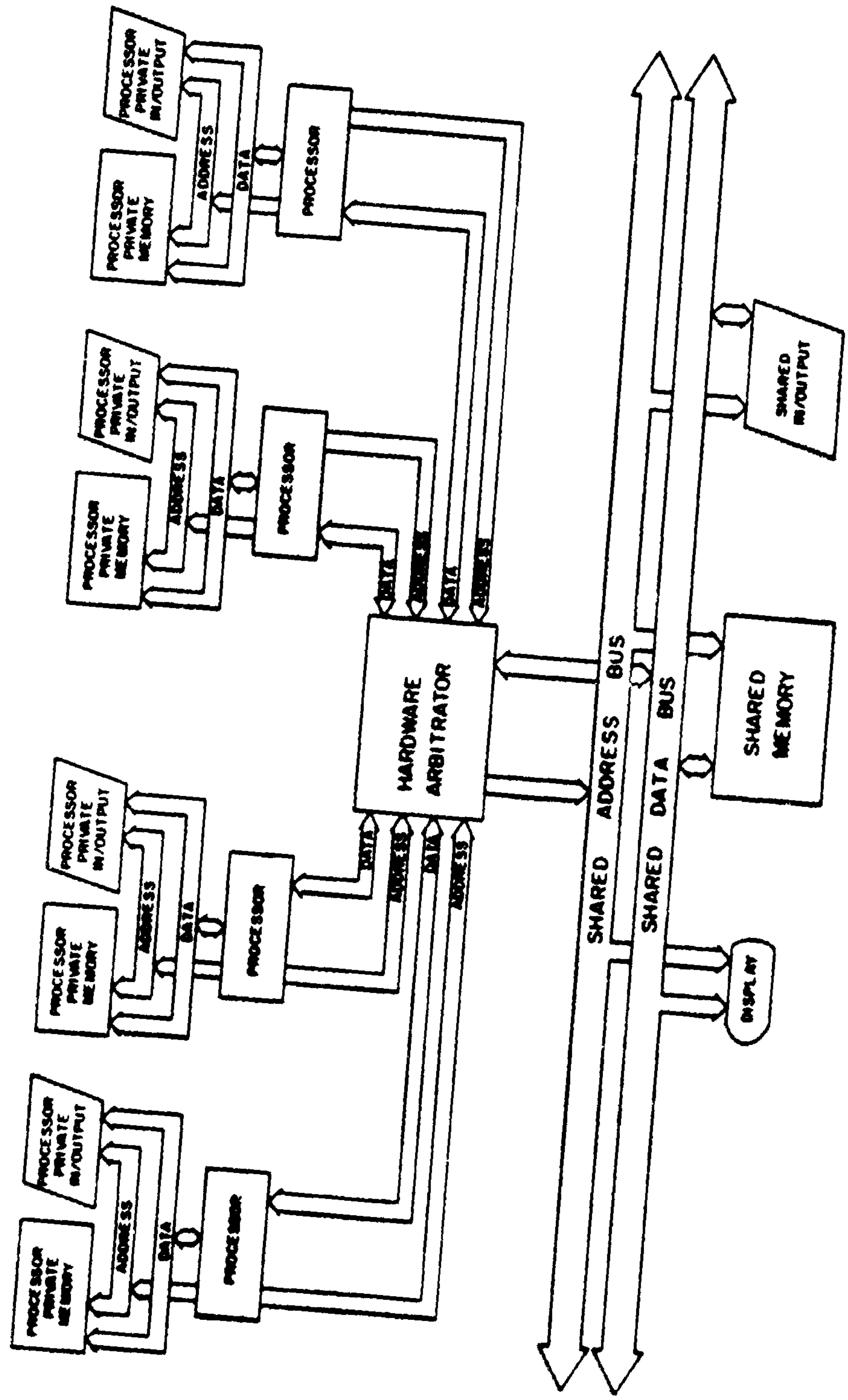

Figure 3.5 Super-65 system block diagram.

and private input/output. This allows the system to be used in many different environments, e.g., single source and single destination, single source and multiple destination, multiple source and single destination, or multiple source and multiple destination.

There is no provision for interprocessor communications other than through Shared Memory. Each processor with its private memory, private input/output and its portion of the Hardware Arbitrator is mounted on a single prototype board which can be plugged directly into one of the seven available peripheral slots on the back of the Apple II. The Hardware Arbitrator consists of tristateable buffers on the Address, Data and Control buses of each of the processors with the required logic and flip-flops to enable the appropriate buffers.

One of the processors (typically the Apple II 6502) is designated as the Controlling Processor (CP). The CP has control over all shared resources. The CP has sole access to both the Shared Address and Control Buses. The CP allows all of the processors access to the Shared Data Bus during a Read from SM. At all other times, the CP has sole access to the Shared Data Bus.

One interesting feature of this architecture is that it requires almost no modification of the Apple II. The reason is that the Apple II 6502 is physically removed from its socket and placed onto a peripheral board. However, a 40-pin conductor connecting the removed 6502 to the empty socket allows the Apple to operate as if the 6502 were actually still in the original socket. The Apple II 6502 then is equivalent to any of the other processors. This allows for a very modular design. That is, every processor board is identical to any other processor board. This simplifies the debugging process a great deal.

One desirable characteristic mentioned previously was that of easy fault detection. A simple type of fault-detection was implemented on the Super-65. This consists of comparing the address on the Shared Address Bus with the current address on each of the processor address bus. If they are not identical, a red LED is lit on the erroneous processor board. Nothing further is done. It is assumed that the operator will observe the problem soon after it occurs and take the proper steps to prevent the faulty processor from contaminating the entire system.

As mentioned earlier, the Apple II uses dynamic RAM and as this memory is divided into 16K by 1 bit chips, it is impossible to treat the lowest 1K of RAM on the Apple differently than the next lowest 15K of RAM. Hence, the Apple II 6502 was placed on a protoboard just like the other processors. In this way all of the processors appear to be peripherals to the Apple II. Because the processors appear as peripherals, whenever one of them becomes the CP, it initiates the equivalent of a DMA. If one examines the schematic diagram of the Apple II (Figure 3.6), one notes that the line $\overline{\text{DMA}}$ disables the buffers attached to the Apple II 6502's address bus. $\overline{\text{DMA}}$ also disables the phase zero clock input to the Apple II 6502. This fact forces one to make one trivial modification to the Apple II in order to allow the Apple II 6502 to operate when it is not the CP. This modification is to disconnect the phase zero input from the AND gate with which $\overline{\text{DMA}}$ is able to disable the phase zero input and connect the phase zero (ϕ_0) to the phase zero (ϕ_0) input on the peripheral connector.

The Apple II 6502 is in control when the Apple II is powered up, just as it would normally be. In order to designate another of the processors as the CP, one simply addresses the memory range \$C*n*xx, where *n* equals the processor number (1, 2, 3, or 7). Seven is the number given to the

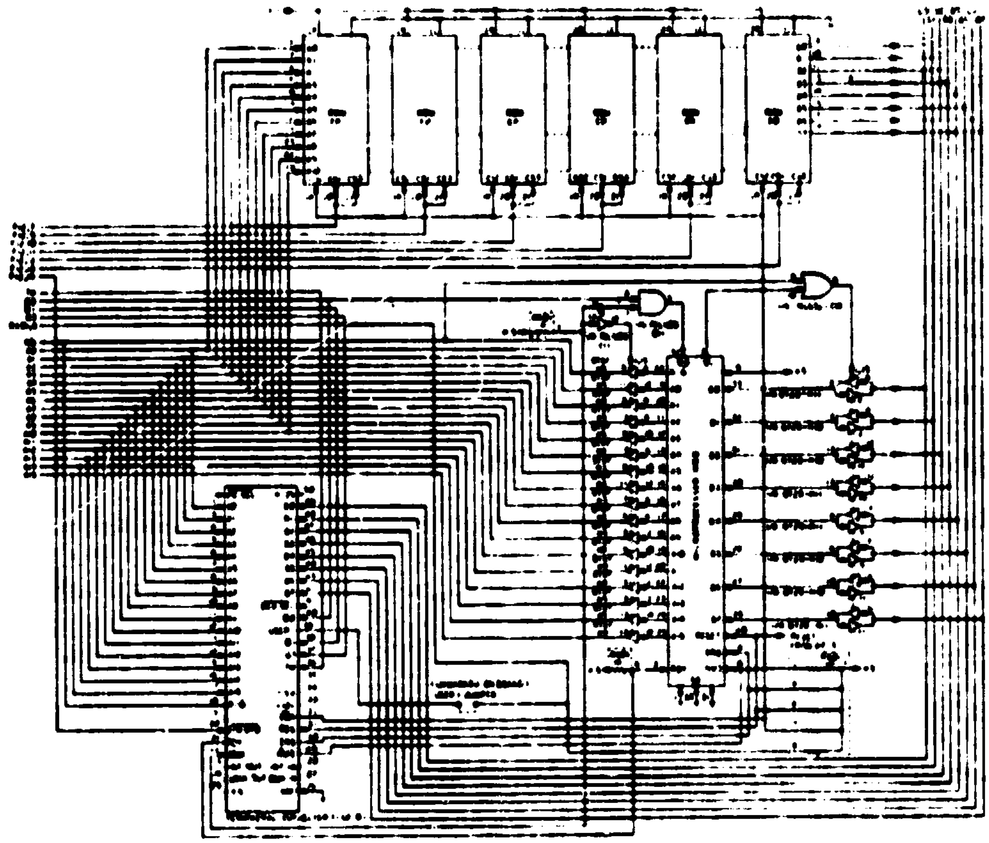

Figure 3.6 Apple II schematic diagram.

Apple II 6502. When one accesses C$n$xx (x = don't care), the Apple II automatically activates the line $\overline{\text{I/O Select}}$ on the nth peripheral slot. This signal is then used to give the processor on the board in slot n control of the shared buses.

For many applications, it is desirable to load the same location in each processors private memory with different values. For example, one may use that location as an indirect pointer, in which case each processor has a different pointer. There are at least three different ways of achieving different values for different processors. One way is to have a hard-wired location in each processor's private memory. This location must have different hard-wired values for each of the processors. One can then manipulate the different values to obtain the desired differences between processors. Another method is to have each of the private memories mapped into the Apple's memory so that the Apple can load different values directly into each of the private memories.

The third method has all of the processors store the value for processor #1 into the location, then disable processor #1, store the value for processor #2 into the same location (writing over the old value), then disable processor #2, and so on until all of the processors have the appropriate pointers. One can then restart all of the processors and proceed with execution of the program. After substantial consideration, it was decided that this method provides sufficient versatility while not requiring as many additional components and board space. Realizing that what was required was one signal that would selectively designate a particular processor for disabling and another signal for restarting all the processors simultaneously, and not wanting to burden the already saturated board with more decoding components, it was decided to use already decoded

signals provided by the Apple II. One recalls that each of the peripheral slots contains three unusual signals; $\overline{\text{I/O Select}}$, $\overline{\text{Device Select}}$ and $\overline{\text{I/O Strobe}}$. $\overline{\text{I/O Select}}$ is used to designate which processor is the CP. As described earlier, $\overline{\text{Device Select}}$ goes low (active) when the address on the Shared Address Bus is in the range \$CON0-\$CONF, where N equals $n + 8$ (n = processor number). This signal then is well-suited for disabling a particular processor. $\overline{\text{I/O Strobe}}$ is contained on all of the peripheral slot connectors and goes low (active) on all of the slots simultaneously, if the address on the Shared Address Bus is in the range \$C800-\$CFFF. Thus, $\overline{\text{I/O Strobe}}$ may be used to restart all the processors simultaneously after the initialization routine.

The design of the Super-65 allows one to expand up to a total of seven processors without any additional hardware being added to the basic system. Furthermore, each processor board is a replica of the previous boards. In this way, one can realistically modify any Apple II to provide it with the capability of a seven processor array without extensive hardware alteration of the Apple II itself. The architecture itself does not limit the number of processors to seven. The Apple II only provides seven available slots with the necessary decoding for the special signals used in this design. If one desired to implement a larger array, one simply needs to provide the extra decoding circuitry and peripheral slots external to the Apple II.

3.4 *Design of the Individual Processor Card*

One recalls from the previous section (Figure 3.5) that the Super-65 architecture provides for Shared Memory (SM), a Hardware Arbitrator, four processors each with its own private memory and input/output capability. As mentioned earlier, the Shared Memory resides on the Apple II main board. The Hardware Arbitrator is distributed across each of the processor cards.

Each processor card contains:

(1) A 6502 microprocessor

(2) A 1K x 8 bit RAM (i.e., 2-2114 memory chips)

(3) Two input/output ports (i.e., 1-6522 VIA)

(4) A tristate buffer for the 6502's address lines (i.e., 2-74LS245)

(5) A tristate buffer for the 6502's data lines

(6) A tristate buffer for certain control signals

(7) Decoding circuitry (i.e., 2-74LS155)

(8) Required logic for implementation of special features (i.e., 74LS00, 74LS04, 74LS05, 74LS30, 74LS74, 74LS76, 74LS85 and 74LS121) (see Figure 3.7).

The power-up procedure resets the 6502, and causes it to jump to a reset vector location. This location is in SM. The enable to the data bus buffer will be active for only two cases: one, during a READ from Shared Memory (this allows all processors to READ simultaneously) and two, during a WRITE operation when the particular processor is the controlling processor (CP). At all other times, the data bus of the 6502 is disconnected from the Shared Data Bus. The enable to the address bus buffers and the READ/$\overline{\text{WRITE}}$ line and $\overline{\text{DMA}}$ line buffers is active only when the specific processor has been designated as the CP. This, then allows a given processor to control the Shared Address Bus, R/$\overline{\text{W}}$ and $\overline{\text{DMA}}$ lines only when that processor is the CP.

As described previously, I/O Select goes low (active) on peripheral connector n when the address on the Shared Address Bus is in the range \$Cn00-\$CnFF (n = processor number). This signal is used to set a flip-flop on the 74LS76 designated as the CP flip-flop. The 'NAND' of the $\overline{\text{I/O Select}}$ signals from all the other peripheral connectors is used to reset the CP

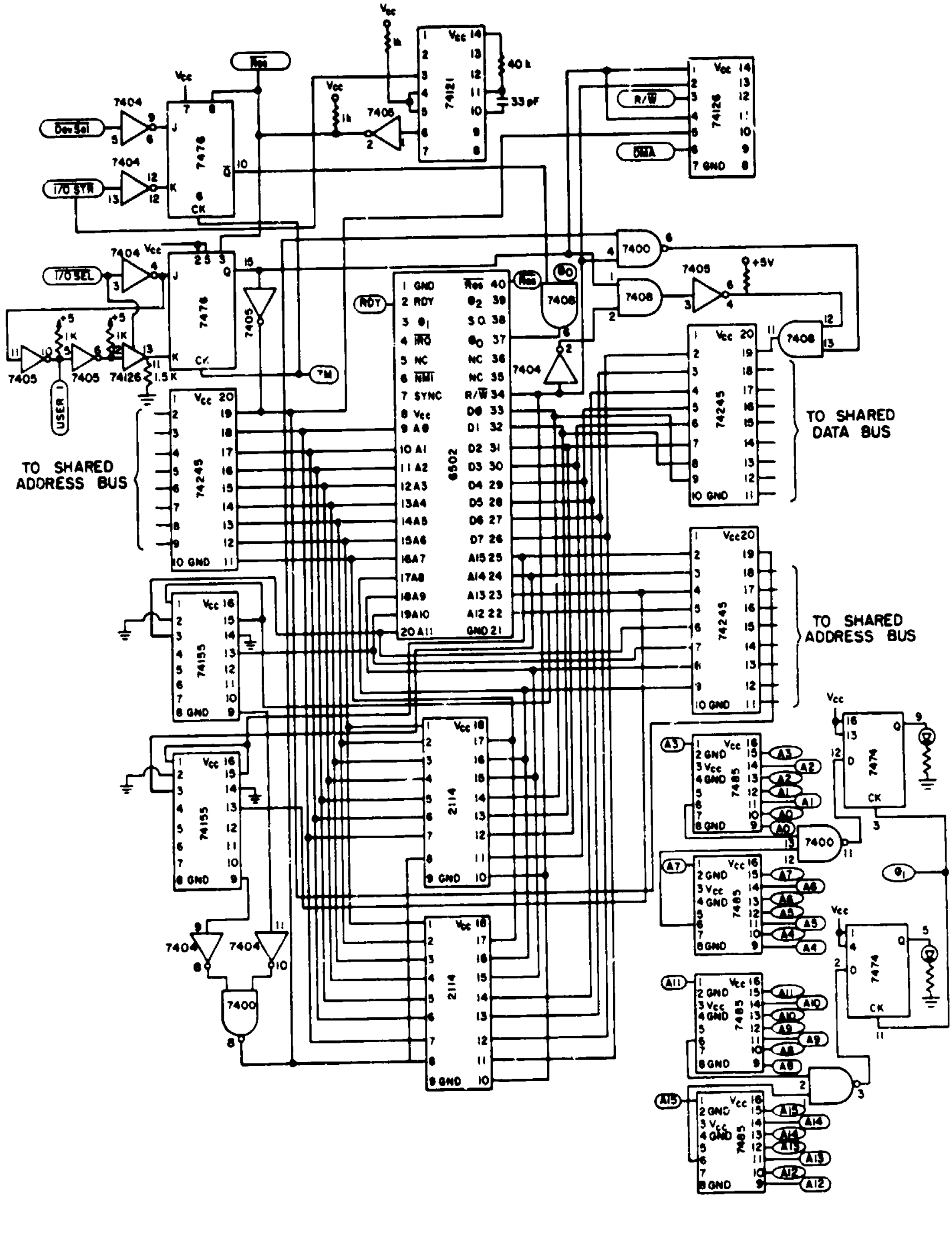

Figure 3.7 Processor card schematic diagram.

flip-flop. Thus, when one of the other processors is designated as CP, the signal which sets its CP flip-flop will simultaneously reset all other CP flip-flops. Since the I/O Select signal remains active for only one-half of a machine cycle, it was necessary to use the 7 MHz signal available on the peripheral connector as the clock input to the CP flip-flop.

The other flip-flop on the 74LS76 is designated as the Disable flip-flop and is used to disable the phase zero (ϕ_0) clock input to the 6502. As described in the preceding section, $\overline{\text{Device Select}}$ is used to disable the desired processor and $\overline{\text{I/O Strobe}}$ is used to restart all the processors simultaneously. Thus, $\overline{\text{Device Select}}$ is used to set the Disable flip-flop and $\overline{\text{I/O Strobe}}$ is used to reset the Disable flip-flop. As the Disable flip-flop output is simply used as a control input to on and gate with the phase zero clock as the other input, resetting the Disable flip-flop would immediately allow the phase zero clock to be applied to the 6502 clock input. Since there is no assurance what the 6502 will do when its clock is removed, the most reliable method of restarting all the processors, is to perform a hardware reset. That is, to pull the $\overline{\text{Reset}}$ line of all the processors low (active) simultaneously. Since the 6502 requires the $\overline{\text{Reset}}$ line remain low for several machine cycles in order to execute a valid reset, it was necessary to have the $\overline{\text{I/O Strobe}}$ line fire a monostable multivibrator (i.e., 74121), that was designed to hold the $\overline{\text{Reset}}$ line low for at least several machine cycles. In order to have sufficient time for all the processors to become synchronized, the monostable was configured to allow a delay of approximately one second. This delay is more than sufficient and could likely be reduced to one millisecond without causing any complications.

Two 74LS155s decode the address lines of the 6502 to determine which

1K of memory the address is making reference to. If the address is in the lowest 1K of memory, the private memory is enabled, else the inverse of the private memory enable is used to designate that a Shared Memory access is requested.

Four 74LS85 (4-bit comparators) are used to compare the address lines of the 6502 with the address lines of the Shared Address Bus. As noted in the Apple II Reference Manual, the address on the address bus becomes valid about 300 nanoseconds after phase one goes high and remains valid through all of phase. Since phase one is the complement of phase zero, when phase zero goes high, phase one goes low. Also, since the address is valid when phase zero goes high and the 74LS74 contains two negative-edge triggered flip-flops, the design compares the upper and lower bytes of the address separately and clocks in the result of the comparators on the falling edge of phase one. Thus, one is able to determine which, if any of the processors is out of step with the others and which byte or bytes of the address is different from that of the CP.

Finally, provision has been made for the inclusion of a 6522 Versatile Interface Adaptor (see Figure 3.8) which contains two 8-bit parallel ports, on each of the processor boards. Implementation of the 6522 will simply be a matter of deciding what address one would like for the I/O ports and various other control words of the 6522 to reside and then providing the necessary decoding circuitry for the 6522. While this is not trivial, it is straightforward and as the main thrust of this work is not actually the hardware implementation, the configuration of the 6522 is left for future research.

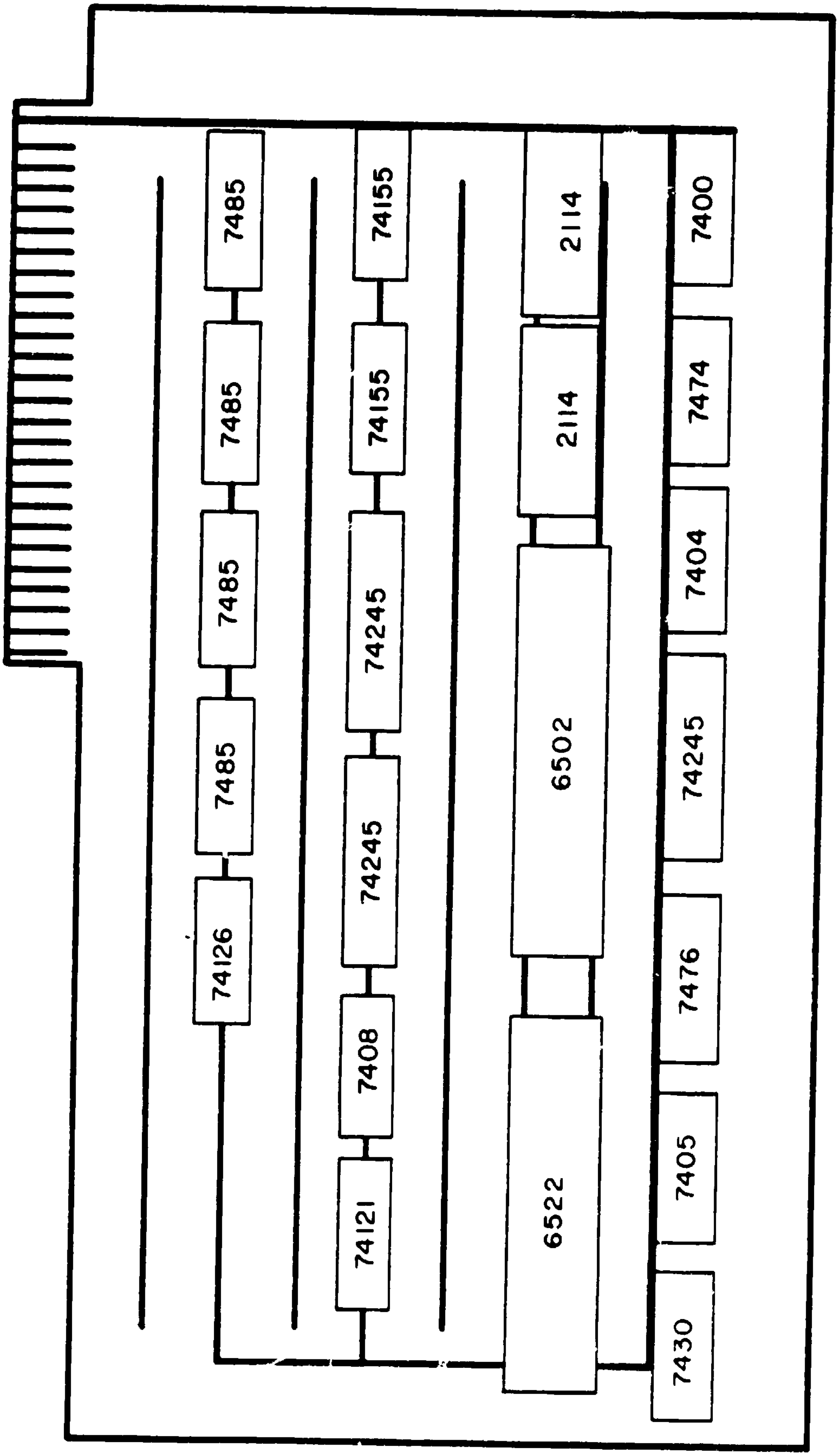

Figure 3.8 Processor card layout.

4. EXAMPLES OF INDEPENDENT DATA HANDLING

4.1 *Introduction*

As described in Section 2.2, independent data handling allows each processor its own source of data. By definition, the data sources are independent of each other. Of course, there are many applications in which the data are not completely independent. However, a less precise but more practical way of differentiating between independent and dependent data handling methods, is to differentiate between applications for which the array is processing a separate problem for each processor (independent data handling) and applications where the array is processing different segments of the same problem for each processor (dependent data handling).

This chapter deals with several examples of independent data programs that are written in Context Independent Code. One will note that independent data programs are usually simpler and more efficient than dependent data programs that perform the same function. This is because independent data programs take advantage of more inherent parallelism and require less overhead of interprocessor communication than dependent data programs.

4.2 *8-Bit Magnitude of Twos-Complement Number*

The following program calculates the magnitude of an 8-bit Twos-Complement Number. This program assumes that each of the processors has its own 8-bit number stored in a location in page zero of private memory designated by the symbolic name, number. The magnitude of the twos complement number stored in location NUMBER is placed in location MAGN at the end of the program.

MAGNITUDE OF TWOS COMPLEMENT NUMBER/CONTEXT INDEPENDENT CODE

```
LABEL   MNEMONIC  OPERAND  CYCLES               COMMENT
BEGIN:    LDA-Z    NUMBER    3;    GET NUMBER
```

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	AND	#80	3;	MASK OFF ALL BUT SIGN BIT
	CIC		2;	
	ROL	ACC	2;	MOVE SIGN BIT INTO LSB POSITION
	ROL	ACC	2;	
	STA-Z	TEMP	3;	STORE SIGN BIT (FOR END AROUND CARRY)
	LDA	#0	2;	
	SEC		2;	
	SBC	TEMP	3;	IF S=0, ACC=0; S=1, ACC=FF
	EOR	NUMBER	3;	IF S=0, ACC=NUMBER, S=1, ACC=$\overline{\text{NUMBER}}$
	CIC		2;	
	ADC	TEMP	3;	ADD END AROUND CARRY
END:	STA-Z	MAGN	3;	
			33;	TOTAL MACHINE CYCLES

To determine the magnitude of a twos complement number, one must decide first if the number is positive or negative. Of course, if the number is positive, one does nothing to it. If the number is negative, one calculates the twos complement inverse of it by complementing it and then adding one to it. The Context Independent Code program stores the sign bit to be used as the end around carry, then subtracts the sign bit from zero to get either 00 or FF. The program then uses the fact that a value exclusive-ored with all zeros is that number (e.g. value = positive) and a value exclusive-ored with all ones is the complement of that value (e.g. value = negative). Finally, the program adds the sign bit (S=0, if positive; S=1, if negative) to obtain the magnitude of the original twos complement number.

4.3 *8 x 8-Bit Multiplication*

The following program multiplies one 8-bit number by another 8-bit number to obtain a 16-bit product. This program assumes that one value is already residing in a location called MPCND and the other value is already present in a location called MPLR, both of which are in page zero of private memory. The product is returned in two locations, PROD-L and PROD-H both of which are in page zero.

8 x 8-BIT MULTIPLICATION/CONTEXT INDEPENDENT CODE

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDA	#00	2;	LOAD IMMEDIATE ZERO
	STA-Z	PROD-L	3;	CLEAR PRODUCT LOW BYTE
	STA-Z	PROD-H	3;	CLEAR PRODUCT HIGH BYTE
	LDX	#08	2;	SET BIT COUNT = 8 BITS
LOOP:	ASL	PROD-L	5;	SHIFT LEFT PRODUCT LOW BYTE
	ROL	PROD-H	5;	ROTATE LEFT PRODUCT HIGH BYTE
	ASL	MPLR	5;	SHIFT LEFT MULTIPLIER
	LDA	#00	2;	SUBTRACT CARRY BIT FROM ZERO TO OBTAIN
	SBC	#00	2;	EITHER 00 (C=1) OR FF (C=0)
	EOR	#FF	2;	COMPLEMENT PREVIOUS RESULT
	AND	MPCND	3;	AND EITHER 00 (CARRY=0) OR FF (CARRY=1)
	STA-Z	TEMP	3;	WITH MULTIPLICAND
	CIC		2;	TEMP = EITHER 00 OR MULTIPLICAND
	ADC	PROD-L	3;	ADD EITHER ZERO OR MULTIPLICAND TO
	STA-Z	PROD-L	3;	SHIFTED PARTIAL PRODUCT LOW BYTE
	LDA-Z	PROD-H	3;	
	ADC	#0	2;	ADD POSSIBLE CARRY TO PRODUCT HIGH BYTE
	STA-Z	PROD-H	3;	
	DEX		2;	DECREMENT BIT COUNT
	BNE	LOOP	2;	DONE? IF NOT, LOOP
END:	RTS		6;	
			392;	TOTAL MACHINE CYCLES REQUIRED

This program uses the algorithm of shifting the partial product left once, then adding the multiplicand to the partial product if the tested bit of the multiplier is set. If the tested bit of the multiplier is zero, the original algorithm would branch around the add instruction and loop back to test the next bit of the multiplier. Because this program is written in Context Independent Code which does not allow conditional branches for which the condition may be different for different processors, instead of branching around the add instruction, this program adds zero when the tested bit of the multiplier is zero.

4.4 *16/8-Bit Binary Division*

The following program is the reverse of the multiplication algorithm. That is, this program takes a 16-bit dividend in two locations called DVND-L and DVND-H and divides them by an 8-bit value stored in a location

called DVSR. All memory locations are assumed to be in page zero of each processors private memory. The 8-bit quotient is returned in a memory location called QNT and the 8-bit remainder is returned in location RMDR.

16/8-BIT DIVISION/CONTEXT INDEPENDENT CODE

```
LABEL   MNEMONIC  OPERAND  CYCLES            COMMENT

BEGIN:   LDX       #08       2;    NUMBER OF BITS IN DIVISOR = 8
         LDA-Z     DVND-L    3;    GET LSB DIVIDEND
         STA-Z     QNT       3;    STORE LSB DIVIDENT IN QUOTIENT
         LDA-Z     DVND-H    3;    GET MSB DIVIDEND
         STA-Z     TEMP      3;
DIVID:   ASL-Z     QNT       5;    SHIFT DIVIDEND-QUOTIENT
         ROL-Z     TEMP      5;    LEFT ONE BIT
         CMP-Z     DVSR      3;    CAN DIVISOR BE SUBTRACTED?
         LDA       #00       2;
         ROL       ACC       2;    GET CARRY BIT INTO LSB OF ACCUMULATOR
         STA-Z     SFLAG     3;    STORE SUBTRACT FLAG BIT
         EOR       #01       2;    COMPLEMENT FLAG BIT, (0=SUBTRACT ZERO
                                     1=SUBTRACT DIVISOR)
         ADC-Z     QNT       3;    INCREMENT QUOTIENT IF DIVISOR COULD BE
                                     SUBTRACTED
         STA-Z     QNT       3;    STORE NEW QUOTIENT
         LDA-Z     SFLAG     3;    GET SUBTRACT FLAG
         ROR       ACC       2;    ROTATE SUBTRACT FLAG BIT TO BORROW
                                     POSITION
         SBC       #00       2;    ACCUMULATOR = (00 IF B=1, FF IF B=0)
         AND-Z     DVSR      3;    ADD DIVISOR WITH EITHER FF OR 00
         STA-Z     SFLAG     3;    STORE EITHER 00 OR DIVISOR
         SEC                 2;
         LDA-Z     TEMP      3;
         SBC-Z     SFLAG     3;    SUBTRACT EITHER 00 OR DIVISOR FROM
                                     DIVIDEND
         DEX                 2;    LOOP UNTIL ALL 8 BITS ARE PROCESSED
         BNE       DIVID     2;
         STA-Z     RMDR      3;    STORE REMAINDER
END:     RTS                 6;
                           447;    TOTAL MACHINE CYCLES REQUIRED
```

This program uses the algorithm of shifting the dividend lift once, then executing a trial subtracting of the divisor. If the subtraction is possible, the quotient is incremented and the actual subtraction executed. As in the multiply program, this program does not use a conditional branch to determine if the subtraction should be done. Instead, the subtract flag determines whether one is subtracting zero or the divisor and whether zero

or one is added to the quotient.

4.5 *32-Bit Accumulation*

The following program accumulates the sum of 255 words, each 32 bits long. The program assumes the data resides in the upper 1020 bytes of the first 1K of memory. The data are stored in four sections with base addresses, BASE-0, BASE-1, BASE-2, and BASE-3. Thus, one is able to access all 255 words by indexed addressing. The five byte result is returned is the first five bytes of page zero called symbolically ACM-0, ACM-1, ACM-2, ACM-3, and ACM-4.

32-BIT ACCUMULATOR/CONTEXT INDEPENDENT CODE

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDA	#00	2;	CLEAR ACCUMULATOR SPACES
	STA-Z	ACM-0	3;	
	STA-Z	ACM-1	3;	
	STA-Z	ACM-2	3;	
	STA-Z	ACM-3	3;	
	STA-Z	ACM-4	3;	
	LDX	#FF	2;	NUMBER OF WORDS = 255
LOOP:	DEX		2;	
	LDA	BASE-0,X	4;	GET LSB
	ADC	ACM-0	3;	ADD TO ACCUMULATOR ZERO
	STA-Z	ACM-0	3;	
	LDA	BASE-1,X	4;	GET NEXT MOST SIGNIFICANT BYTE
	ADC	ACM-1	3;	ADD TO ACCUMULATOR ONE
	STA-Z	ACM-1	3;	
	LDA	BASE-2,X	4;	GET NEXT MOST SIGNIFICANT BYTE
	ADC	ACM-2	3;	ADD TO ACCUMULATOR TWO
	STA-Z	ACM-2	3;	
	LDA	BASE-3,X	4;	GET MSB
	ADC	ACM-3	3;	ADD TO ACCUMULATOR THREE
	LDA	#00	2;	
	ADC	ACM-4	3;	ADD POSSIBLE CARRY TO ACCUMULATOR FOUR
	STA-Z	ACM-4	3;	
	CPX	#00	2;	
	BNE	LOOP	2;	ALL 255 NUMBERS ADDED?
	RTS		6;	
			16,857;	TOTAL MACHINE CYCLES REQUIRED

This program is very close to a standard 6502 32-bit accumulation program. The only possible change would be to only add the carry to ACM-4

when is was set. That is, to branch around the add instruction when the carry was zero. However, later this program with independent data will be compared to the same application with dependent data. That is, instead of performing a different 32-bit accumulation for each of four processors, one performs a single 32-bit accumulation using all of the four processors.

4.6 *32 x 32-Bit Binary Multiplication*

The following program multiplies one 32-bit number by another 32-bit number to obtain a 64-bit product. The program assumes that the multiplicand already resides in four bytes called symbolically MPCD-0, MPCD-1, MPCD-2 and MPCD-3. The program also assumes that the multiplier already resides in four bytes called symbolically MPLR-0, MPLR-1, MPLR-2 and MPLR-3. The 64-bit product is returned in eight bytes called symbolically PRD-0, PRD-1, PRD-2, PRD-3, PRD-4, PRD-5, PRD-6 and PRD-7. All memory locations are assumed to be in page zero.

32 x 32-BIT MULTIPLICATION/CONTEXT INDEPENDENT CODE

```
LABEL   MNEMONIC  OPERAND  CYCLES           COMMENT

BEGIN:   LDA       #00      2;   CLEAR PRODUCT BYTES 0-7
         STA-Z     PRD-0    3;
         STA-Z     PRD-1    3;
         STA-Z     PRD-2    3;
         STA-Z     PRD-3    3;
         STA-Z     PRD-4    3;
         STA-Z     PRD-5    3;
         STA-Z     PRD-6    3;
         STA-Z     PRD-7    3;
         LDX       #20      2;   32 BITS IN MULTIPLIER
SHIFT:   ASL       PRD-0    5;   SHIFT PRODUCT BYTES 0-7 LEFT ONE BIT
         ROL       PRD-1    5;
         ROL       PRD-2    5;
         ROL       PRD-3    5;
         ROL       PRD-4    5;
         ROL       PRD-5    5;
         ROL       PRD-6    5;
         ROL       PRD-7    5;
         ASL       MPLR-0   5;   SHIFT NEXT BIT OF MULTIPLIER INTO CARRY
         ROL       MPLR-1   5;     POSITION
         ROL       MPLR-2   5;
```

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	ROL	MPLR-3	5;	
	LDA	#00	2;	
	SBC	#00	2;	
	EOR	#FF	2;	IF C=0, ACCUM=00, IF C=1, ACCUM=FF
	STA-Z	MASK	3;	STORE MASK
	AND	MPCD-0	3;	IF C=0 MASK OFF MULTIPLICAND BYTE 0
	STA-Z	TMP-0	3;	STORE EITHER MPCD-0 OR 00
	LDA-Z	MASK	3;	
	AND	MPCD-1	3;	
	STA-Z	TMP-1	3;	STORE EITHER MPCD-1 OR 00
	SDA-Z	MASK	3;	
	AND	MPCD-2	3;	
	STA-Z	TMP-2	3;	STORE EITHER MPCD-2 OR 00
	LDA-Z	MASK	3;	
	AND	MPCD-3	3;	
	STA-Z	TMP-3	3;	STORE EITHER MPCD-3 OR 00
	CIC		2;	
	LDA-Z	PRD-0	3;	
	ADC	TMP-0	3;	ADD EITHER 00 OR MPDC-0 TO PRODUCT
	STA-Z	PRD-0	3;	
	LDA-Z	PRD-1	3;	
	ADC	TMP-1	3;	ADD EITHER 00 OR MPCD-1 TO PRODUCT
	STA-Z	PRD-1	3;	
	LDA-Z	PRD-2	3;	
	ADC	TMP-2	3;	ADD EITHER 00 OR MPCD-2 TO PRODUCT
	STA-Z	PRD-2	3;	
	LDA-Z	PRD-3	3;	
	ADC	TMP-3	3;	ADD EITHER 00 OR MPCD-3 TO PRODUCT
	STA-Z	PRD-3	3;	
	LDA-Z	PRD-4	3;	
	ADC	#00	2;	ADD POSSIBLE CARRY
	STA-Z	PRD-4	3;	
	LDA-Z	PRD-5	3;	
	ADC	#00	2;	ADD POSSIBLE CARRY
	STA-Z	PRD-5	3;	
	LDA-Z	PRD-6	3;	
	ADC	#00	2;	ADD POSSIBLE CARRY
	STA-Z	PRD-6	3;	
	LDA-Z	PRD-7	3;	
	ADC	#00	2;	ADD POSSIBLE CARRY
	STA-Z	PRD-7	3;	
	DEX		2;	ALL 32 BITS PROCESSED?
	BNE	SHIFT	2;	
END:	RTS		6;	
			5506;	TOTAL MACHINE CYCLES REQUIRED

This program simply expands the single byte multiplication program in Context Independent Code to that of a four byte multiplication. This requires shifting of a product which is eight bytes rather than two, and

requires much more temporary storage but is a straightforward extension of the simpler program.

4.7 *Comparison of CIC Programs With Uniprocessor Programs*

The following program calculates the magnitude of an 8-bit twos complement number. This is a standard uniprocessor program and hence is not in CIC.

8-BIT MAGNITUDE/UNIPROCESSOR CODE

```
LABEL  MNEMONIC  OPERAND  CYCLES              COMMENT

BEGIN:  LDA-Z     NUMBER     3;    GET NUMBER
        BPL       END        2;    IF POSITIVE, DONE
        EOR       #FF        2;    INVERT NEGATIVE NUMBER
        CIC                  2;
        ADC       #01        2;    ADD END AROUND CARRY
DONE:   STA-Z     MAGN       3;
                            14;    CYCLES IF NEGATIVE
                             8;    CYCLES IF POSITIVE
```

The uniprocessor program requires approximately 11 machine cycles on the average to obtain the magnitude of an 8-bit twos complement number. The CIC program, on the other hand, always requires 33 machine cycles to do the same job. Thus, one must use 3 processors to obtain the same throughput as the original processor for this task. If one were to represent the throughput of the array as Kn times the single processor throughput where n = number of processors and K is the 'recoding factor', the recoding factor for the 8-bit twos complement magnitude program is 0.33. This simply means that for this task, the number of processors used should always be greater than three for effective use of CIC programming.

The following is the uniprocessor program from *Leventhal* [1979] that calculates the 16-bit product of two 8-bit numbers.

8 x 8 MULTIPLICATION/UNIPROCESSOR

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDA	00	2;	LSB OF PRODUCT = ZERO
	STA-Z	PRD-H	3;	MSB OF PRODUCT = ZERO
	LDX	08	2;	8 BITS IN MULTIPLIER
SHIFT:	ASL	ACC	2;	SHIFT PRODUCT LEFT ONE BIT
	ROL	PRD-H	5;	
	ASL	MPLR	5;	SHIFT MULTIPLIER LEFT ONE BIT
	BCC	NO ADD	2;	NO ADDITION IF NEXT BIT IS ZERO
	CIC		2;	
	ADC	MPCD	3;	ADD MULTIPLICAND TO PARTIAL PRODUCT
	BCC	NO ADD	2;	
	INC	PRD-H	5;	ADD CARRY TO MSB IF PRODUCT
NO ADD:	DEX		2;	
	BNE	SHIFT	2;	LOOP UNTIL 8 BITS ARE MULTIPLIED
	STA-Z	PRD-L	3;	STORE LSBS OF PRODUCT
	RTS		6;	
			208;	MACHINE CYCLES TYPICALLY REQUIRED

Since the CIC program requires 392 machine cycles, the recoding factor for this program is 0.53. Effective use of CIC programming requires that one employ a number of processors that is greater than two. One should note that the uniprocessor program requires only one accumulator for efficient processing. For example, the uniprocessor does not initially clear the LSB of the product, nor does it store the result of the addition within the loop. Also, the LSB of the product is left in the accumulator, which allows it to be shifted much more quickly than if it were in page zero. All these facts allow the uniprocessor program to be executed much quicker than the CIC program. If one were to have another accumulator available (as in the 6800), the difference would be reduced to replacing the BCC instruction with LDA 00, SBC 00, EOR FF, and MPCND and STA-Z TEMP. The additional execution time would then be approximately 80 machine cycles or about thirty-eight percent longer. As one can see, the effectiveness of CIC programming depends as much on the expertise of the programmer as any other single factor.

The following is a uniprocessor 16/8-bit division program.

16/8-BIT DIVISION/UNIPROCESSOR

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDX	#08	2;	DIVISION BITS = 8
	LDA-Z	DVND-L	3;	GET LSB DIVIDEND
	STA-Z	QNT	3;	
	LDA-Z	DVND-H	3;	GET MSB DIVIDEND
DIVID:	ASL-Z	QNT	5;	SHIFT DIVIDEND-QUOTIENT LEFT ONE BIT
	ROL	ACC	2;	
	CMP-Z	DVSR	3;	CAN DIVISOR BE SUBTRACTED?
	BCC	NO SUB	2;	NO, GO TO NEXT STEP
	SBC	DVSR	3;	YES, SUBTRACT DIVISOR AND INCREMENT
	INC	QNT	5;	QUOTIENT LOOP UNTIL ALL 8 BITS ARE
NO SUB:	DEX		2;	DIVIDED
	BNE	DIVID	2;	
	STA-Z	RMDR	3;	STORE REMAINDER
END:	RTS		6;	
			200;	TYPICAL MACHINE CYCLES REQUIRED

The CIC program requires 447 machine cycles and thus the recoding factor for this program is 0.45. In this case, three processors are required to obtain a throughput greater than the throughput of a single processor executing uniprocessor code.

The uniprocessor 32-bit accumulation program is identical to the CIC program except that instead of adding a possible carry to the fifth byte, one inserts a BCC instruction which causes one not to execute the add if the carry is not set. This will reduce the execution time by 2.5 machine cycles on the average. For all practical considerations the CIC program executes as fast as the uniprocessor program and hence K = 1.0.

The following is a uniprocessor 32 x 32-bit multiplication program.

32 X 32-BIT MULTIPLICATION/UNIPROCESSOR

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDA	#00	2;	
	STA-Z	PRD-0	3;	CLEAR PRODUCT BYTES 0-7
	STA-Z	PRD-1	3;	
	STA-Z	PRD-2	3;	
	STA-Z	PRD-3	3;	

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	STA-Z	PRD-4	3;	
	STA-Z	PRD-5	3;	
	STA-Z	PRD-6	3;	
	STA-Z	PRD-7	3;	
	LDX	#20	2;	32-BITS IN MULTIPLIER
SHIFT:	ASL	PRD-0	5;	SHIFT PRODUCT BYTES 0-7 LEFT 1 BIT
	ROL	PRD-1	5;	
	ROL	PRD-2	5;	
	ROL	PRD-3	5;	
	ROL	PRD-4	5;	
	ROL	PRD-5	5;	
	ROL	PRD-6	5;	
	ROL	PRD-7	5;	
	ASL	MPLR-0	5;	SHIFT NEXT BIT OF MULTIPLIER INTO CARRY
	ROL	MPLR-1	5;	POSITION
	ROL	MPLR-2	5;	
	ROL	MPLR-3	5;	
	BCC	NO ADD	2;	NO ADDITION IF NEXT BIT IS ZERO
	CIC		2;	
	LDA-Z	PRD-0	3;	CARRY SET, ADD MULTIPLICAND TO PARTIAL
	ADC	MPCD-0	3;	PRODUCT
	STA-Z	PRD-0	3;	
	LDA-Z	PRD-1	3;	
	ADC	MPCD-1	3;	
	STA-Z	PRD-1	3;	
	LDA-Z	PRD-2	3;	
	ADC	MPDC-2	3;	
	STA-Z	PRD-2	3;	
	LDA-Z	PRD-3	3;	
	ADC	MPCD-3	3;	
	STA-Z	PRD-3	3;	
	BCC	NO ADD	2;	
	LDA-Z	PRD-4	3;	
	ADC	#00	2;	
	STA-Z	PRD-4	3;	
	LDA-Z	PRD-5	3;	
	ADC	#00	2;	
	STA-Z	PRD-5	3;	
	LDA-Z	PRD-6	3;	
	ADC	#00	2;	
	STA-Z	PRD-6	3;	
	LDA-Z	PRD-7	3;	
	ADC	#00	2;	
	STA-Z	PRD-7	3;	
NO ADD	DEX		2;	
	BNE	SHIFT	2;	ALL 32 BITS MULTIPLIED
END:	RTS		6;	
			4450;	TOTAL CYCLES FOR ALL BITS SET
			2032;	TOTAL CYCLES FOR ALL BITS CLEARED

On the average, the required machine cycles might be near the arithmetic average of the two extremes or approximately 3266. Since the CIC program requires 5506 machine cycles for execution. The recoding factor for this program is approximately 0.59. Thus, two processors executing the CIC program would yield a greater throughput than a single processor executing the uniprocessor program.

To summarize, the 8-bit magnitude CIC program has $K = 0.33$. The 8 x 8-bit multiplication program written in CIC has $K = 0.53$. The 16/8-bit division program in CIC has $K = 0.45$. The 32-bit accumulator program in CIC has $K = 1.0$ and finally the 32 x 32-bit multiplication program has $K = 0.59$. The important thing to remember is that once the price has been paid by writing the program in CIC, one can gain throughput linearly with additional processors. The experimental evidence here shows that the recoding factor, K, varies from a maximum of 1.0 (32-bit accumulation) to a minimum of approximately 0.45 (16/8-bit division). Thus, if an array of processors were to execute these sample programs, one would see a throughput somewhere between $0.45n$ and n times the throughput of a single processor. This indicates that each PE has an efficiency of at least 45% for these sample programs.

5. EXAMPLES OF DEPENDENT DATA HANDLING

5.1 *Introduction*

This chapter exhibits different examples of dependent data handling. Applications of dependent data handling must have sufficient inherent parallelism within them to allow each of the processors to process a different segment of the entire problem. For this reason, dependent data problems are typcially larger and more complex than independent data problems. It is quite possible that a dependent data problem will have independent data subroutines used within it. This is because the calculation for which the subroutine is used does not require knowledge that the data for each of the processors is related in some way. There are two different methods of dependent data handling. The first method employs each of the 8-bit processors to process 8-bit data and communicate to some of the other processors certain results of its processing. The other method uses all of the n processors to process $8n$-bit data spread across all of the processors. This method needs a higher degree of communication between the processors than the other method as the processors are being used to simulate a single more powerful processor with a word size of $8n$-bits.

5.2 *Carry-Propagation Problem*

As suggested in the preceding section, when one teams up several processors to simulate a larger processor several obstacles appear. One of the most difficult to resolve is that of a carry being propagated from one processor to another. In multiple precision arithmetic, a carry out of the most significant bit position is placed into what is called the carry bit. This bit is then added to the least significant bit of the next word. This works well when one is using a single processor. However, when one is using several processors, the carry from the most significant bit of one

processor should be added to the least significant bit of the next processor. One problem is that the carry-out is not available on an external pin (except bit-slice microprocessors) and there exists no carry in pin so that one could join general-purpose microprocessors together in much the same manner as digital systems designers have previously joined several adders together to form a large adder. Even if the pins did exist, one would encounter a carry propagation problem similar to that encountered by digital designers. The solution in that case was to use carry look ahead adder cells. Another solution to this carry propagation problem is discussed in the following section.

5.3 *Stored-Carry Solution*

The preceding section described the carry propagation problem and examined a possible hardware solution to the problem other than redesign of the microprocessor. One solution is to transfer the carry from each of the processors to the SM and then transfer the proper carry to the next processor. This solution is inefficient because every addition requires at least one WRITE to SM and one READ from SM.

The original solution of transferring each carry through SM is modified so that it is necessary to transfer one word containing several carries to the next processor only after 255 additions. This 'Stored Carry' method assumes that a large number of values are to be added. Each processor performs double precision arithmetic in adding 255 values together. The upper byte then contains the carries from the last 255 additions (a maximum of 255). One then transfers the stored carry word to the next processor through SM and adds it to that processor's accumulated value. This method requires only one WRITE to and READ from SM for 255 additions and is therefore much more efficient than the other method.

5.4 *32-Bit Accumulation*

The following program calculates the sum of 255 thirty-two-bit words. The program assumes that six locations in page zero have been previously initialized to the following values.

LOCATION	PROCESSOR 0	PROCESSOR 1	PROCESSOR 2	PROCESSOR 3
FA	00	01	02	03
FB	00	00	00	00
FC	00	00	00	00
FD	B0	B1	B2	B3
FE	00	00	00	00
FF	C7	C1	C2	C3

32-BIT ACCUMULATION/CIC (DEPENDENT DATA)

```
LABEL   MNEMONIC  OPERAND  CYCLES              COMMENT

BEGIN:   LDA       #00      2;
         STA-Z     SUM      3;   CLEAR SUM LOCATION
         STA-Z     CARRY    3;   CLEAR STORED CARRY BYTE
         LDX       #FF      2;
LOOP:    DEX                2;
         LDA       BASE,X   4;   GET NEXT VALUE TO BE ADDED TO SUM
         CIC                2;
         ADC       SUM      3;
         STA-Z     SUM      3;
         LDA       #00      2;
         ADC       CARRY    3;
         STA-Z     CARRY    3;
         CPX       #00      2;   ALL 255 VALUES SUMMED?
         BNE       LOOP     2;   IF NOT, LOOP
         LDA-Z     CARRY    3;   LOAD ALL STORED CARRY WORDS
         STA       (FC),X   6;   STORE CARRY WORD FROM CPU-0
         INX                2;
         STA       $C100    4;   SET CP=1
         STA       (FC),X   6;   STORE CARRY WORD FROM CPU-1
         INX                2;
         STA       $C200    4;   SET CP=2
         STA       (FC),X   6;   STORE CARRY WORD FROM CPU-2
         LDA       $B000    4;   GET CARRY WORD FROM CPU-0
         STA-Z     $0001    3;   STORE CARRY FOR CPU-1
         LDA       $B100    4;   GET CARRY WORD FROM CPU-1
         STA-Z     $0002    3;   STORE CARRY FOR CPU-2
         LDA       $B200    4;   GET CARRY WORD FROM CPU-2
         STA-Z     $0003    3;   STORE CARRY FOR CPU-3
         STA       $C700    4;   SET CP=0
         LDX       #00      2;
         STX       $0000    3;   CLEAR CARRY WORD FOR CPU-0
```

```
LABEL  MNEMONIC  OPERAND  CYCLES              COMMENT

         LDA     (FA),X     6;    LOAD ALL CARRIES SIMULTANEOUSLY
         CIC                2;
         ADC      SUM       3;    ADD CARRIES TO SUM
        STA-Z     SUM       3;    STORE RESULT
         STA     (FC),X     6;    TRANSFER SUM BYTE FROM CPU-0
         STA     $C100      4;    CP=1
         STA     (FC),X     6;    TRANSFER SUM BYTE FROM CPU-1
         STA     $C200      4;    CP=2
         STA     (FC),X     6;    TRANSFER SUM BYTE FROM CPU-2
         STA     $C300      4;    CP=3
         STA     (FC),X     6;    TRANSFER SUM BYTE FROM CPU-3
        LDA-Z    CARRY      3;
         ADC      #00       2;    ADD POSSIBLE CARRY TO CPU-3s CARRY WORD
         INX                5;
         STA     (FC),X     6;    STORE MSB OF ACCUMULATION
                         7034;    TOTAL MACHINE CYCLES REQUIRED
```

The preceding program performs 16-bit addition in accumulating 255 eight-bit words for each processor. Each processor then has a sum byte and a carry byte. The carry bytes are then transferred through the SM to the next processor. The carry word from the previous processor is then added to the sum byte of the current processor. Finally, each of the sum bytes is transferred to the SM with CPU-3 transferring both the sum byte and the carry byte in order to complete the 5 bytes necessary to accumulate 255 thirty-two-bit numbers. One notes that it is necessary to perform four WRITEs to and READS from SM in order to transfer the carry word from each processor to the next processor. This is due to the architecture chosen which allows interprocessor communication only through SM. The process of transferring carry words to each of the processors and transferring the results to SM takes 139 machine cycles. If the architecture permitted the transfers to be done in one pass instead of four, the transfers would only have taken 35 machine cycles. This represents a savings of 104 of a total of 7034 machine cycles required for the accumulation, only 1.5 percent. However, one should not lose sight of the potential problem when the number

of processors becomes large. If one does not want the amount of time required for transferring data between processors to take more than 10 percent of the entire program, one cannot use this architecture for more than approximately twenty processors. That is because the transfer requires $35n$ machine cycles and the entire program takes approximately 7000 machine cycles. Thus, for n greater than twenty, the transfer alone will require more than 10 percent of the array's time. However, this result assumes than n-byte arithmetic is used and is is unlikely that one would ever require 20-byte precision.

5.5 *32 x 32-Bit Multiplication*

This program multiplies one 32-bit number by another 32-bit number to obtain a 64-bit product. The multiplicand is assumed to be located in four bytes called D0, D1, D2 and D3 situated in page zero for all of the processors. The four bytes of the multiplier are distributed among each of the processors. That is, CPU-0 has R0, CPU-1 has R1, CPU-2 has R2 and CPU-3 has R3. When the program refers to R, it is referring to the respective byte of the multiplier which each processor has. The program has each processor multiply its multiplier byte by the low byte of the multiplicand. This product is placed into two locations called S0 and S1. The multiplier byte is then multiplied by the second byte of the multiplicand. The low byte of this 16-bit product is added to S1 and the high byte is placed in S2. The third byte of the multiplicand is multiplied by the multiplier byte. The low byte of this product is added to S2 and the high byte is placed in S3. The fourth byte of the multiplicand is multiplied by the multiplier byte. The low byte of this product is added to S3 and the high byte of the product is placed in S4. Each processor then transfers its S0-S4 words to SM. The partial sums S0-S4 are stored in a shifted manner

to indicate their weightings (Figures 5.1a and 5.1b). All the partial sums are read into each processor's page zero. Then the X index register of each processor is initialized to a different value so that each processor indexes to different partial sums for accumulation. For example, CPU-0 adds its S0 to zero and stores the result in Lo-Accum. It then adds its S1 to CPU-1' S1 with carry and stores this in Mid-Accum. CPU-0 adds zero to Lo-Accum and to Mid-Accum twice more to get the final result of Lo-Accum and Mid-Accum. In order to account for possible carries from one processor to another, zero is added with carry to a null location called Hi-Accum. Of course, while CPU-0 is doing this, the other processors are accumulating their own Lo-Accum, Mid-Accum and Hi-Accum from their own indexed data. The only thing left to do is to transfer the stored carry Hi-Accum byte to the next processor and add it in to obtain the final 64-bit product. This program assumes the same initialization as the 32-bit accumulator program.

32 x 32-BIT MULTIPLY/CIC (DEPENDENT DATA)

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
BEGIN:	LDX	#7	2;	
	LDA	#ZERO	2;	
	STA-Z	E8	3;	INITIALIZE BASE POINTER ZERO
	LDA	#ONE	2;	
	STA-Z	EA	3;	INITIALIZE BASE POINTER ONE
	LDA	#TWO	2;	
	STA-Z	EC	3;	INITIALIZE BASE POINTER TWO
	LDA	#THREE	2;	
	STA-Z	EE	3;	INITIALIZE BASE POINTER THREE
	LDA	#00	2;	
	STA	(E8),X	6;	CLEAR NULL LOCATIONS
	STA	(EA),X	6;	
	STA	(EC),X	6;	
	DEX		2;	
	STA	(E8),X	6;	
	STA	(EA),X	6;	
	DEX		2;	
	STA	(E8),X	6;	
	LDX	#2	2;	
	STA	(EE),X	6;	
	DEX		2;	

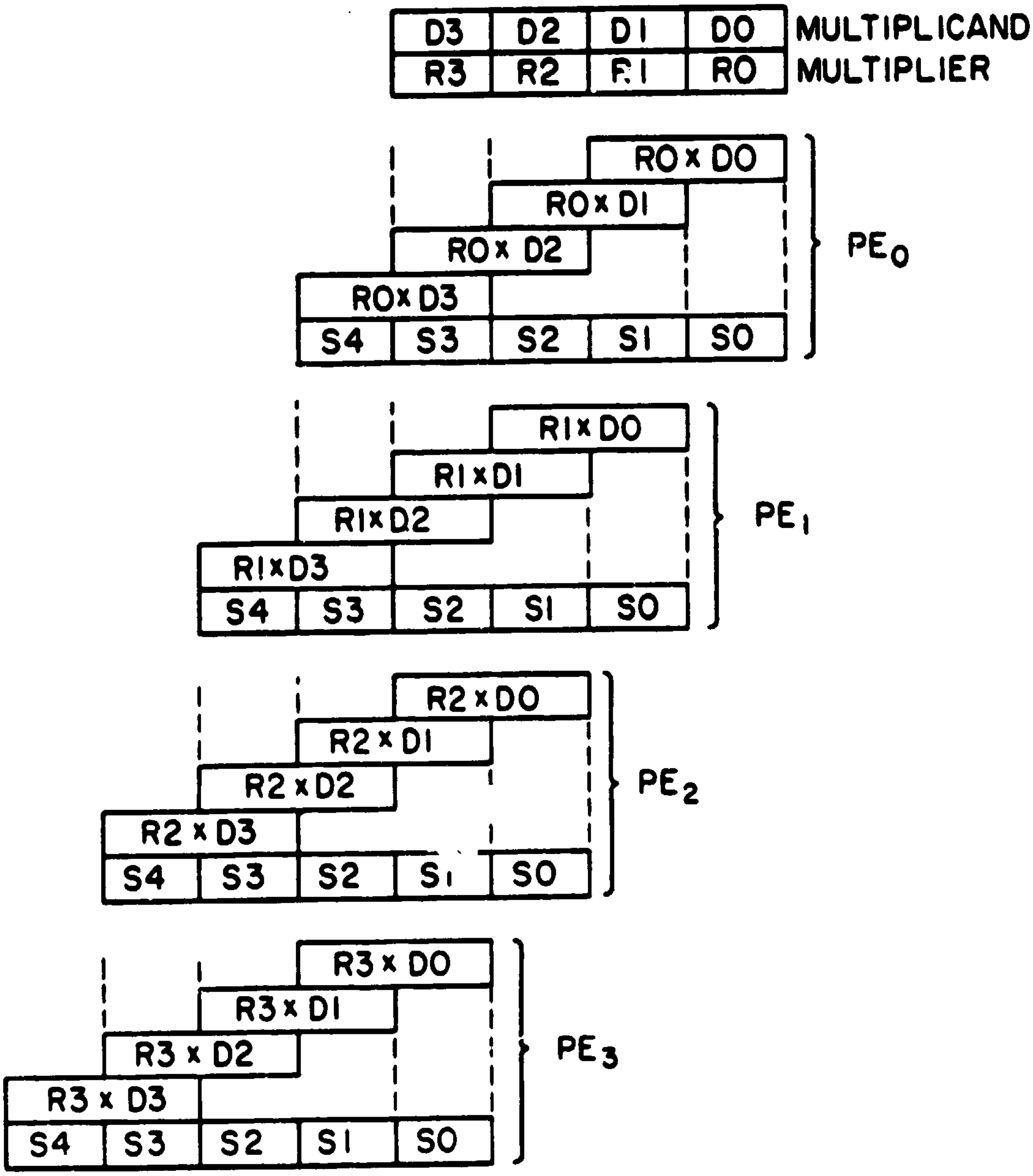

Figure 5.1a 32 x 32-bit multiplication diagram.

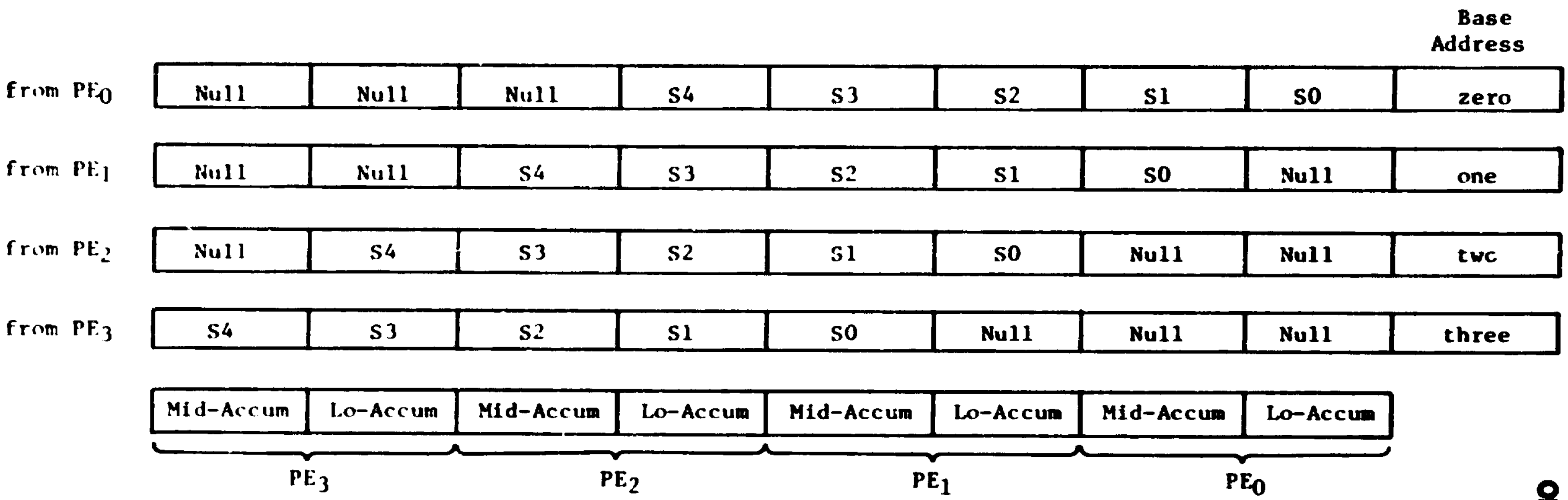

Figure 5.1b 32 x 32-bit multiplication diagram.

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	STA	(EE),X	6;	
	STA	(EC),X	6;	
	DEX		2;	
	STA	(EE),X	6;	
	STA	(EC),X	6;	
	STA	(EA),X	6;	
	LDA-Z	R	3;	GET MULTIPLIER BYTE
	STA-Z	MPLR	3;	PASS PARAM. TO 8-BIT MULTIPLY SUBROUTINE
	LDA-Z	D0	3;	GET LOW BYTE OF MULTIPLICAND
	STA-Z	MPCD	3;	PASS PARAM. TO 8-BIT MULTIPLY SUBROUTINE
	JSR	8-BIT MULTIPLY	398;	DO 8-BIT MULTIPLY OF R AND D0
	LDA-Z	PRD-L	3;	
	STA-Z	S0	3;	PLACE LOW BYTE OF PRODUCT INTO S0
	LDA-Z	PRD-H	3;	
	STA-Z	S1	3;	PLACE HIGH BYTE OF PRODUCT INTO S1
	LDA-Z	D1	3;	GET 2ND BYTE OF MULTIPLICAND
	STA-Z	MPCD	3;	PASS PARAM. TO 8-BIT MULTIPLY SUBROUTINE
	JSR	8-BIT MULTIPLY	398;	DO 8-BIT MULTIPLY OF R AND D1
	CIC		2;	
	LDA-Z	PRD-L	3;	
	ADC	S1	3;	ADD LOW BYTE OF R X D1 TO S1
	STA-Z	S1	3;	
	LDA	#00	2;	
	ROR	ACC.	2;	
	STA-Z	CARRY	3;	STORE POSSIBLE CARRY
	LDA-Z	PRD-H	3;	
	STA-Z	S2	3;	
	LDA-Z	D2	3;	GET 3RD BYTE OF MULTIPLICAND
	STA-Z	MPCD	3;	PASS PARAM. TO 8-BIT MULTIPLY SUBROUTINE
	JSR	8-BIT MULTIPLY	398;	DO 8-BIT MULTIPLY OF R AND D2
	LDA-Z	CARRY	3;	
	ASL	ACC.	2;	PLACE POSSIBLE CARRY BACK
	LDA-Z	PRD-L	3;	GET LOW BYTE OF PRODUCT R X D2
	ADC	S2	3;	
	STA-Z	S2	3;	ADD LOW BYTE OF R X D2 WITH CARRY TO S2
	LDA	#00	2;	
	ROR	ACC.	2;	
	STA-Z	CARRY	3;	STORE POSSIBLE CARRY
	LDA-Z	PRD-H	3;	
	STA-Z	S3	3;	STORE HIGH BYTE OF R X D2 IS S3
	LDA-Z	D3	3;	GET 4TH BYTE OF MULTIPLICAND
	STA-Z	MPCD	3;	PASS PARAM. TO 8-BIT MULTIPLY SUBROUTINE
	JSR	8-BIT MULTIPLY	398;	DO 8-BIT MULTIPLY OF R AND D3
	LDA-Z	CARRY	3;	
	ASL	ACC.	2;	RETURN CARRY FROM LAST ADD
	LDA-Z	PRD-L	3;	
	ADC	S3	3;	ADD LOW BYTE OF R X D3 TO S3

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	STA-Z	S3	3;	STORE FINAL S3
	LDA-Z	PRD-H	3;	
	ADC	#00	2;	ADD IN POSSIBLE CARRY
	STA-Z	S4	3;	STORE FINAL S4
	LDX	#04	2;	
	STA	(FC),X	6;	TRANSFER S4 FROM CPU-0
	DEX		2;	
	LDA-Z	S3	3;	
	STA	(FC),X	6;	TRANSFER S3 FROM CPU-0
	DEX		2;	
	LDA-Z	S2	3;	
	STA	(FC),X	6;	TRANSFER S2 FROM CPU-0
	DEX		2;	
	LDA-Z	S1	3;	
	STA	(FC),X	6;	TRANSFER S1 FROM CPU-0
	DEX		2;	
	LDA-Z	S0	3;	
	STA	(FC),X	6;	TRANSFER S0 FROM CPU-0
	STA	$C100	4;	SET CP=1
	LDX	#05	2;	
	LDA-Z	S4	3;	
	STA	(FC),X	6;	TRANSFER S4 FROM CPU-1
	DEX		2;	
	LDA-Z	S3	3;	
	STA	(FC),X	6;	TRANSFER S3 FROM CPU-1
	DEX		2;	
	LDA-Z	S2	3;	
	STA	(FC),X	6;	TRANSFER S2 FROM CPU-1
	DEX		2;	
	LDA-Z	S1	3;	
	STA	(FC),X	6;	TRANSFER S1 FROM CPU-1
	DEX		2;	
	LDA-Z	S0	3;	
	STA	(FC),X	6;	TRANSFER S0 FROM CPU-1
	STA	$C200	4;	SET CP=2
	LDX	#06	2;	
	LDA-Z	S4	3;	
	STA	(FC),X	6;	TRANSFER S4 FROM CPU-2
	DEX		2;	
	LDA-Z	S3	3;	
	STA	(FC),X	6;	TRANSFER S3 FROM CPU-2
	DEX		2	
	LDA-Z	S2	3;	
	STA	(FC),X	6;	TRANSFER S2 FROM CPU-2
	DEX		2;	
	LDA-Z	S1	3;	
	STA	(FC),X	6;	TRANSFER S1 FROM CPU-2
	DEX		2;	
	LDA-Z	S0	3;	
	STA	(FC),X	6;	TRANSFER S0 FROM CPU-2
	STA	$C300	4;	SET CP=3

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	LDX	#07	2;	
	LDA-Z	S4	3;	
	STA	(FC),X	6;	TRANSFER S4 FROM CPU-3
	DEX		2;	
	LDA-Z	S3	3;	
	STA	(FC),X	6;	TRANSFER S3 FROM CPU-3
	DEX		2;	
	LDA-Z	S2	3;	
	STA	(FC),X	6;	TRANSFER S2 FROM CPU-3
	DEX		2;	
	LDA-Z	S1	3;	
	STA	(FC),X	6;	TRANSFER S1 FROM CPU-3
	DEX		2;	
	LDA-Z	S0	3;	
	STA	(FC),X	6;	TRANSFER S0 FROM CPU-3
	STA	$C700	4;	SET CP=0
	LDX	#05	2;	
QLOOP:	DEX		2;	GET S0-S4 FROM CPU-0 FOR ALL CPUs
	LDA	(FC),X	6;	
	STA	(E8),X	6;	
	CPX	#00	2;	
	BNE	QLOOP	2;	
	STA	$C100	4;	SET CP=1
	LDX	#06	2;	
RLOOP:	DEX		2;	TRANSFER S0-S4 FROM CPU-1 TO ALL CPUs
	LDA	(FC),X	6;	
	STA	(EA),X	6;	
	CPX	#01	2;	
	BNE	RLOOP	2;	
	STA	$C200	4;	SET CP=2
	LDX	#07	2;	
TLOOP:	DEX		2;	TRANSFER S0-S4 FROM CPU-2 TO ALL CPUs
	LDA	(FC),X	6;	
	STA	(EC),X	6;	
	CPX	#02	2;	
	BNE	TLOOP	2;	
	STA	$C300	4;	SET CP=3
	LDX	#08	2;	
ULOOP:	DEX		2;	TRANSFER S0-S4 FROM CPU-3 TO ALL CPUs
	LDA	(FC),X	6;	
	STA	(EE),X	6;	
	CPX	#03	2;	
	BNE	ULOOP	2;	
	LDA-Z	$FA	3;	GET INDEX REG X=0 FOR CPU-0, 2 FOR CPU-1
	ASL	ACC.	2;	4 FOR CPU-2, 6 FOR CPU-3
	STX		2;	
	LDA	(E8),X	6;	ACCUMULATE 3 BYTES FOR EACH CPU, THAT IS,
	CIC		2;	LO-ACCUM, MID-ACCUM AND HI-ACCUM
	ADC	(EA),X	6;	HI-ACCUM = STORED CARRIES
	STA-Z	LO-ACCUM	3;	
	INX		2;	

LABEL	MNEMONIC	OPERAND	CYCLES	COMMENT
	LDA	(E8),X	6;	
	ADC	(EA),X	6;	
	STA-Z	MID-ACCUM	3;	
	LDA	#00	2;	
	ADC	#00	2;	
	STA-Z	HI-ACCUM	3;	
	CIC		2;	
	LDA-Z	LO-ACCUM	3;	
	DEX		2;	
	ADC	(EC),X	6;	
	STA-Z	LO-ACCUM	3;	
	INX		2;	
	LDA-Z	MID-ACCUM	3;	
	ADC	(EC),X	6;	
	STA-Z	MID-ACCUM	3;	
	LDA-Z	HI-ACCUM	3;	
	ADC	#00	2;	
	STA-Z	HI-ACCUM	3;	
	CIC		2;	
	LDA-Z	LO-ACCUM	3;	
	DEX		2;	
	ADC	(EE),X	6;	
	STA-Z	LO-ACCUM	3;	FINISH ACCUMULATION OF 3 BYTE SUMS
	INX		2;	
	LDA-Z	MID-ACCUM	3;	
	ADC	(EE),X	6;	
	STA-Z	MID-ACCUM	3;	
	LDA-Z	HI-ACCUM	3;	
	ADC	#00	2;	
	STA-Z	HI-ACCUM	3;	
	DEX		2;	
	LDY	#00	2;	
	STA	(FC),Y	6;	TRANSFER STORED CARRIES FROM CPU-0
	STA	$C100	4;	SET CP=1
	STA	(FC),Y	6;	TRANSFER STORED CARRIES FROM CPU-1
	STA	$C200	4;	SET CP=2
	STA	(FC),Y	6;	TRANSFER STORED CARRIES FROM CPU-2
	STA	$C700	4;	SET CP=0
	LDA	#00	2;	
	STA	(E8),Y	6;	CLEAR STORED CARRY WORD FOR CPU-0
	LDA	(FC),Y	6;	GET STORED CARRY WORD FOR CPU-1
	LDY	#02	2;	
	STA	(E8),Y	6;	GIVE S.C. WORD FROM CPU-0 TO ALL CPUs
	STA	$C100	4;	SET CP=1
	LDY	#00	2;	
	LDA	(FC),Y	6;	GET STORED CARRY WORD FROM CPU-1
	LDY	#04	2;	
	STA	(E8),Y	6;	GIVE S.C. WORD FROM CPU-1 TO ALL CPUs
	STA	$C200	4;	SET CP=2
	LDY	#00	2;	
	LDA	(FC),Y	6;	GET S.C. WORD FROM CPU-2

```
LABEL  MNEMONIC  OPERAND   CYCLES              COMMENT

       LDY       #06          2;
       STA       (E8),Y       6;    GIVE S.C. WORD FROM CPU-2 TO ALL CPUs
       CIC                    2;
       LDA-Z     LO-ACCUM     3;
       ADC       (E8),X       6;    ADD IN STORED CARRIES
       STA-Z     LO-ACCUM     3;
       LDA-Z     MID-ACCUM    3;
       ADC       #00          2;
       STA       MID-ACCUM    3;
                           2685;    TOTAL MACHINE CYCLES REQUIRED
```

From the above program, one can easily see how various independent data programs may be placed within a larger and more complex dependent data program. One example is the 8-bit multiply independent data subroutine which is used in the 32 x 32-bit multiply dependent data program below.

5.6 *Comparison of CIC Programs With Uniprocessor Programs*

The 32-bit accumulation (dependent data) program requires all four processors, but is able to execute an accumulation of 255 thirty-two bit numbers in 7034 machine cycles whereas the uniprocessor program requires 16,857 machine cycles to do the same job. Thus, the four-processor array can perform the 32-bit accumulation is 2.4 times as fast as the single processor. This represents a recoding factor of 0.60. These results would certainly encourage one to pursue array processing. An important fact to point out is that with the architecture used in this study, propagating carries from one processor to the next for 8n-bit precision arithmetic, will take a larger and larger amount of time as the number of processors grows. However, if the architecture were modified to allow all the processors to propagate their carries in one pass instead of n passes, this would not be the case. For this array of 4 processors, propagating the carries required less than 2 percent of the execution time, therefore, the architecture did not significantly hamper throughput. With K = 0.60, each of

the microprocessors is about 60 percent efficient. So long as carry-propagation delay is not significant, one could expect a throughput of $0.6n$ for an array of n processors executing this program.

The 32 x 32-bit multiply (dependent data) program requires all four processors but is able to perform a 32 x 32-bit multiply in 2685 machine cycle as compared to the average execution time of 3266 for the uniprocessor program. Thus, one is able to obtain a speed-up of 1.22 by using four processors. To determine the recoding factor for this program, one calculates the throughput of the array as Kn. In this case, the recoding factor K is roughly 0.30. If one examines this program closely, it is apparent that almost 20 percent of the time is spent in transferring values from one processor through SM to another processor. Thus, the architecture used to implement the four-processor array is seriously hampering the efficiency of this program by requiring four times as long to pass parameters between processors. If one were to implement another architecture which would allow all the parameters to be passed in a single sweep, the recoding factor, K would reach approximately 0.40. In this case, an array of processors could be expected to exhibit a throughput of roughly $0.4n$.

6. SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

6.1 *Summary*

As stated earlier, the main objective of this work was to assert that the recoding of standard uniprocessor programs into Context Independent Code programs is feasible for an important set of applications. This objective was achieved by implementing a four-processor array and recoding several programs for it. The programs were divided into two categories, independent and dependent data handling. The first category allowed each of the processors to work on a separate set of data such that no processor required any results from any other processor in order to complete its task. The second category required each of the processors to work on a subset of the entire problem. This meant that the processors needed to communicate intermediate results with one another at various times in order to complete their task.

In order to allow easier comprehension of the programs, the design of the Super-65 array processor was described in detail and alternative approaches were analyzed. The strengths and weaknesses of the implementation chosen for the Super-65 were noted. Among the strengths were: its simplicity, both in processor-memory interconnection and interprocessor connections, expandability of the system, no host processor required and the fault tolerance potential of the design. Among the weaknesses of the design were: its restriction of interprocessor communications, the fact that only one processor is able to write to SM at a time and the fact that all programs executed on the Super-65 had to be written in Context Independent Code. The last weakness does not inhibit this study at all as the intent of this study is to examine the implementation of Context Independent Code. The first two weaknesses of the design did not affect

the independent data programs nearly as much as the dependent data programs. The effect of the specific architecture on the efficiency of the CIC programs was noted in the case of the dependent data programs and a possible alternative architecture was suggested.

The conclusion drawn from the independent data programs was that for the set of recoded programs, if one were to have an array of n processors, the throughput of the array would range somewhere between $0.45n$ and n times the throughput of a single processor. This means that the recoding factor of the sample programs ranged from 0.45 to 1.00. This is, of course, quite impressive and encouraging.

The conclusion drawn from the dependent data programs was that for the set of recoded programs, if one had an array of n processors, the throughput of the array would range between $0.3n$ and $0.6n$ times the throughput of a single processor. As expected, the dependent data programs were considerably less efficient than the independent data programs with the recoding factor for the sample programs from 30 to 60 percent (see Table 6.1).

It is thus apparent that the throughput of the array processor is highly dependent on the type of programs that it is executing. However, if one considers that it has been shown by this study to be possible to obtain a linear relationship between throughput and the number of processors in the array (so long as carry-propagation delay in minimal), one must admit that Context Independent Code may provide the key to arrays of immense proportions. One may conclude that this study has shown that the implementation of Context Independent Code is not only feasible for array programs, but is in fact desirable as it allows the array throughput to be linearly related to the array size. Limitations to the array size are not due to

TABLE 6.1 Recoding factors for the sample program

	Independent Data	Dependent Data
8-Bit Magnitude	0.33	
8 x 8-Bit Multiply	0.53	
16/8-Bit Divide	0.45	
32-Bit Accumulator	1.00	0.60
32 x 32-Bit Multiply	0.59	0.30

the CIC program but rather are due to the hardware restraints that one chooses to impose. Of course, if the array were to be infinitely large, the time delay from one end of the array to the other could become significant. Context Independent Code further has the property that the processors never become unsynchronized once they are initialized because all the processors are always forced to execute the exact same instruction. That is, none of the processors is allowed to be turned off during the execution of a CIC program.

6.2 *The Ideal Microprocessor for an Array of Microprocessors*

It is not attempted here to define the ideal microprocessor for an array of microprocessors. Instead several desirable qualities that are found lacking in the microprocessor used for the Super-65 are described, as well as those features of the 6502 microprocessor which are extremely useful will be noted as well.

The most vital feature of the 6502 is its indirect indexed addressing mode. This feature allows the processors to execute the same instruction but locally index the effective address so that the processors actually access different memory locations at the same time. This property is essential as it allows one the ability to use pointers to point to the desired data locations. Also it allows one to index through data from a base location. Since a READ from SM is always executed by all of the processors values read from SM are stored in the same locations in all processors. One way to allow different processors to obtain different data while executing the same instruction is to use indirect indexed addressing where either the indirect value or the index value is a local value.

Another important mode of addressing is Indexed Indirect Addressing where one can index through a table of pointers for different data. This

mode is not quite as useful as the previous mode but still provides the programmer a much more versatile set of instuctions.

One of the most distinctive features of the 6502 is its Zero-Page Addressing mode. This mode allows the programmer to access any of the 256 locations of page zero very rapidly and thus allows one to use page zero in the same manner as a small cache memory. This addressing mode allows for considerable increase in throughput of the 6502 if used efficiently. However, for many applications, 256 locations are insufficient to contain all the necessary data and for three cases, Zero-Page Addressing is not as attractive as it could be. A modified Zero-Page Addressing mode may be much more useful for larger programs. This modified Zero-Page Addressing mode can be called Designated Page Addressing. This mode requires an 8-bit page register that can be set to any of the 256 different pages in the 6502 memory. In this way, one can designate which page of memory is desired to have fast access. This allows the microprocessor to execute at almost twice its regular speed as it would seldom be necessary to specify the high byte of each address. One executes a 'Set Page' instruction at the beginning of the program and then execute the bulk of the instructions from that page in memory. If it becomes necessary to cross into the next page or some other page of memory for a considerable number of instructions or data, one simply sets the page to a different number. Another benefit is that for stack-oriented code, the designated page may be set to that page of memory where the stack resides. This could allow one to access the stack very quickly for non-stack operations. Altogether, this designated page option is strongly recommended.

One characteristic of the 6502 is that one cannot do memory-to-memory manipulations. That is, one must always route one of the operands by way

of the accumulator. This does not allow one to keep any temporary result in the accumulator and also forces the programmer to use more instructions to perform memory-to-memory transfer. For this reason, another accumulator may be desirable, particularly one which has the full capabilities of the original accumulator. This accumulator might be transparent to the programmer such that the microprocessor is capable of memory-to-memory manipulation without passing through an accumulator.

Placing a microprocessor into an array system, especially the Super-65 means extensive use of the index registers. More such registers, preferably with general-purpose register capabilities of shifting, incrementing, decrementing and the like could be used effectively. The addition of at least one general purpose register with the option of adding the contents of that register to the accumulator may resolve the temporary storage problem.

In contrast to the MC6800, the 6502 does not have tristate capability on the chip. That is, the 6502 does not itself provide DMA capability. However, the architecture of the Super-65 could not have taken advantage of this capability had it been available. This is because each processor should always be able to access its private memory. This would not be the case if the tristate buffers for the address and data buses were on the microprocessor chip. If the microprocessor has 512 bytes of RAM on-board and tristate buffers on-board with control inputs to determine when the buffers should be tristated, one could reduce chip count on the processor board significantly. This reduction might not be worth the required additional complexity of the microprocessor chip. However, with the onset of VLSI, the above option might be easily within reach.

Most microprocessors have the capability of being halted for varying

amounts of time. This is typically done by either a HALT signal or by disabling the clock input to the microprocessor. When the clock is disabled, the most reliable procedure is to reset the microprocessors before proceeding. It is desirable to temporarily cause the microprocessor to execute no-ops with the clock active so that the microprocessor remains synchronized with the other processors of the array. Thus, the capability to disable instruction decoding within the microprocessor and force execution of no-ops until the instruction decode disable control input goes inactive would be quite useful in deselecting certain processors for a few instructions.

One final property that present microprocessors do not have is the ability to team the processors easily to do multi-word arithmetic as a single unit. In particular, there is no method of propagating carries from one processor to the next without loading the accumulator with zero and shifting the carry bit into the accumulator, then storing the accumulator where the next processor can read it. This fact led to implementation of the stored-carry solution. However, the stored-carry solution works reasonably well when several additions are necessary. When only one addition is required the stored-carry approach is extremely inefficient and unsatisfactory. One solution would be to place carry in and carry out pins on each microprocessor. This solution would lead to lengthy carry-propagation delays which would be unacceptable. A possible alternative would be to place carry-propagate and carry-generate inputs and outputs on each processor. This method would require two additional pins but would allow the carry propagation delay time to be substantially smaller than the preceding solution.

6.3 *Extending the Microprocessor Array*

There are several obstacles to extending the microprocessor array to a very large number. These obstacles are due to the hardware implementation of the Super-65 rather than the implementation of Context Independent Code.

The most serious impairment is that for an array of n processors, n WRITES to SM and n READS from SM are required to pass information from each processor to the next. In the four-processor array implemented, this problem was not conspicuous. One can readily see that for a larger array, the percentage of time spent simply communicating between processors could rapidly become unacceptable. Therefore, in order to extend the array substantially, one should modify the interprocessor communications to allow each processor to communicate at least to its nearest neighbors by performing a single WRITE. This can be done perhaps most easily by giving each processor two special locations within its private memory. Whenever the processor WRITES to one of these locations, it is giving information to one of its two nearest neighbors. Whenever the processor READS from one of the two special locations it is receiving information from one of its two nearest neighbors. This would relieve the processor communication bottleneck.

A related problem is that the implemented array requires that each processor wait its turn to store its results in SM. Once again, this forces the array to perform n times as many writes as the uniprocessor would normally do. It is true that two processors cannot WRITE onto the same address and data buses at the same time. A possible solution might be to have all the processors store their word into a special location in Private Memory that is part of a special piece of hardware. This hardware would be designed to accept the address from the controlling processor and store each processor's word in a sequential fashion beginning at the

address specified by the CP. This hardware would, of course, need to operate significantly faster than the processors. For very large arrays, it might be necessary to follow every WRITE to SM with one or two no-ops to allow the hardware time to complete the transfer of every processor's word. This then would reduce the store time to SM from n WRITE instructions to 1 WRITE instruction and possibly 1 or 2 no-op instructions.

The design of the processor board was meant to allow implementation of an arbitrarily large array. Except for one detail, this was achieved. The original design of the processor board includes an 8-input NAND gate to be attached to the reset input of the control processor flip-flop on each board. Obviously, this will not allow one to have more than 8 other processors or a total of 9 processors. In order to remedy this situation, open-collector buffers are placed on each of the inputs normally tied to the NAND gate. The wired-AND of these inputs is formed by tying them together to pin 39 of the peripheral connector. User 1 must be disabled on the Apple. Finally the NAND of the inputs (equivalent to the previous design) is achieved by attaching the wired-AND to the input of an inverter. The output of the inverter is then tied to the reset input through a tri-stateable buffer whose control input is I/O Select. The buffer prevents I/O Select from first setting its CP flip-flop and then resetting it immediately afterwards. This modified design does not of itself limit the array size.

One final restriction is that the Apple II backplane has space for only 7 processor boards, and in order to simplify the decoding, the Apple II devotes an entire page to each I/O Select line, each Device Select line and it devotes 8 pages to I/O Strobe. Since the Apple II provides only 7 slots with I/O Select, Device Select and I/O Strobe, it is not trivially

possible to extend the array size. However, there is a commercially available card cage with the desired number of slots, required power supply and required decoding for the size of array desired. One would probably devote only 1 location to each I/O Select, Device Select and I/O Strobe control signal, allowing the array size to reach several thousand.

6.4 *Suggestions for Further Research*

The first suggestion for further research is to correct the imperfections within the Super-65 design that have been previously noted. Specifically, one should provide:

(1) more sophisticated interprocessor communication,

(2) some method of storing in SM more rapidly,

(3) necessary decoding circuitry, etc. to allow expansion of the array.

Then one should pursue the recoding of many more programs into Context Independent Code. In particular, one should determine if it is possible to recode dependent data programs in such a way so as not to spend an unacceptable percentage of the time transferring results from one processor to another. One should try to more fully determine the restrictions to CIC programming and if possible develop more well-defined rules for implementing it.

Other areas for extended research include pursuing the design of the ideal microprocessor for an array environment. One could determine if:

(1) 512 bytes of on-board RAM

(2) Tri-state address and data buffers on board

(3) Designated page option

(4) Instruction decoding disable control input

(5) Extra accumulator

C-2

(6) Additional in' sgisters

(7) Carry propagate/generate inputs and outputs

are all within the practical reach of today's technology, and if so, what sacrifices would be necessary in order to achieve all of the above options.

Finally, one might wish to review all the previously mentioned issues and try to determine what, if any impact the use of a 16-bit microprocessor would have upon them.

REFERENCES

Artwick, B. A. [1980], *Microcomputer Interfacing,* Prentice-Hall Inc., Erglewood Cliffs, CA.

Barn..., G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick and R. A. Stokes, [1968], The ILLIAC IV Computer, *IEEE Trans. on Computers, C-17(8)*, 746-757.

Barnwell, T. P., III, S. Gaglio and R. M. Price [1978], A Multi-Microprocessor Architecture for Digital Signal Processing, *Proc. of the International Conference on Parallel Processing,* 115-119.

Borden, W., Jr. [1978], *The Z-80 Microcomputer Handbook,* Howard W. Sams & Co. Inc., Indianapolis, IN.

Camp, R. C., T. A. Smay and C. J. Triska [1979], *Microprocessor Systems Engineering,* Matrix Publishing Inc.

Espinosa, C. [1979], *The Apple II Reference Manual,* Apple Computer Inc., Cupertino, CA.

Flynn, M. J. [1972], Some Computer Organizations and Their Effectiveness, *IEEE Trans. on Computers, C-21(9),* 948-960.

Garland, H. [1979], *Introduction to Microprocessor System Design,* McGraw-Hill, New York, NY.

Kuck, D. J. [1968], ILLIAC IV Software and Application Programming, *IEEE Trans. on Computers C-17(8)*, 758-770.

Leventhal, L. A. [1979], *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Inc., Berkeley, CA.

Machado, N. C. [1972], An Array Processor With a Large Number of Processing Elements, Ph.D. Thesis, Dept. Computer Science, Univ. of Ill., Urbana-Champaign, IL.

Sawin, D. H. [1977], *Microprocessors and Microcomputer Systems*, D. C. Heath, and Co., Lexington, MA.

Scanlon, L. J. [1980], *6502 Software Design*, Howard W. Sams & Co. Inc., Indianapolis, IN.

Slotnick, D. L. [1967], Unconventional Systems, *Proc. AFIPS Spring Joint Computer Conference*, 477-481.

Thurber, K. J. and L. D. Wald [1975], Associative and Parallel Processors, *Computing Surveys*, *7(4)*, 215-255.

APPENDIX I

IMPLEMENTATION OF THE 8-BIT MULTIPLICATION ROUTINE

This appendix is presented in order to document the 8 x 8 bit multiplication program that was demonstrated on the Super-65. The initialization routine takes the values stored in locations $1001 and $1002 and places them in locations $0062 and $0063 respectively, within each PEM. PE is then disabled and the contents of locations $1003 and $1004 are transferred to locations $0062 and $0063 within each PEM. PE is then disabled and the contents of location $1005 and $1006 are transferred to locations $0062 and $0063 within each PEM. PE is then disabled and the contents of location $1007 and $1008 are transferred to locations $0062 and $0063 within PEM . All PEs are then restarted and the array performs a RESET to synchronize the PEs. This initialization routine places the different multipliers into location $0062 and the different multiplicands into location $0063 of each PEM. If each PE had its own I/O port, it could read its own multiplier and multiplicand from that port and the initialization routine just described would not be required.

8 x 8-BIT MULTIPLICATION INITIALIZATION ROUTINE

LOCATION	OBJECT CODE	MNEMONIC	OPERAND	N	COMMENT
3900	AD 01 10	LDA	$1001	4;	GET MULTIPLIER FOR PE
3903	85 62	STA	$ 62	3;	STORE IN PEMs
3905	AD 02 10	LDA	$1002	4;	GET MULTIPLICAND FOR PE
3908	85 63	STA	$ 63	3;	STORE IN PEMs
390A	8D 90 C0	STA	$C090	4;	DISABLE PE
390D	AD 03 10	LDA	$1003	4;	GET MULTIPLIER FOR PE
3910	85 62	STA	$ 62	3;	STORE IN PEMs
3912	AD 04 10	LDA	$1004	4;	GET MULTIPLICAND FOR PE
3915	85 63	STA	$ 63	3;	STORE IN PEMs
3917	8D B0 C0	STA	$C0B0	4;	DISABLE PE
391A	AD 05 10	LDA	$1005	4;	GET MULTIPLIER FOR PE
391D	85 62	STA	$ 62	3;	STORE IN PEMs
391F	AD 06 10	LDA	$1006	4;	GET MULTIPLICAND FOR PE
3922	85 63	STA	$ 63	3;	STORE IN PEMs

LOCATION	OBJECT CODE	MNEMONIC	OPERAND	N	COMMENT
3924	8D D0 C0	STA	$C0D0	4;	DISABLE PE
3927	AD 07 10	LDA	$1007	4;	GET MULTIPLIER FOR PE
392A	85 62	STA	$ 62	3;	STORE IN PEM
392C	AD 08 10	LDA	$1008	4;	GET MULTIPLICAND FOR PE
392F	85 63	STA	$ 63	3;	STORE IN PEM
3931	8D FF CF	STA	$CFFF	4;	RESTART ALL PEs
				72;	TOTAL MACHINE CYCLES REQUIRED

The following program assumes that locations $0062 and $0063 have previously been loaded with the multiplier and multiplicand respectively. Location $0064 is used as a temporary storage location and locations $0060 and $0061 contain the 16-bit product (low and high bytes respectively), after execution of the program.

8x8-BIT MULTIPLICATION/INDEPENDENT DATA

LOCATION	OBJECT CODE	MNEMONIC	OPERAND	N	COMMENT
4000	A9 00	LDA	00	2;	LOAD IMMEDIATE ZERO
4002	85 60	STA	$60	3;	CLEAR PRODUCT LOW BYTE
4004	85 61	STA	$61	3;	CLEAR PRODUCT HIGH BYTE
4006	A2 08	LDX	08	2;	SET BIT COUNT = 8 BITS
4008	06 60	ASL	$60	5;	SHIFT LEFT PRODUCT LOW BYTE
400A	26 61	ROL	$61	5;	ROTATE LEFT PRODUCT HIGH BYTE
400C	06 62	ASL	$62	5;	SHIFT LEFT MULTIPLIER
400E	A9 00	LDA	00	2;	SUBTRACT CARRY BIT FROM ZERO TO
4010	E9 00	SBC	00	2;	OBTAIN EITHER 00 (C=1) OR FF (C=0)
4012	49 FF	EOR	FF	2;	COMPLEMENT PREVIOUS RESULT
4014	25 63	AND	$63	3;	AND EITHER 00 (C=0) OR FF (C=1)
4016	85 64	STA	$64	3;	WITH MULTIPLICAND
4018	18	CLC		2;	TEMP = EITHER 00 OR MULTIPLICAND
4019	65 60	ADC	$60	3;	ADD EITHER ZERO OR MULTIPLICAND TO
401B	85 60	STA	$60	3;	SHIFTED PARTIAL PRODUCT LOW BYTE
401D	A5 61	LDA	$61	3;	
401F	69 00	ADC	00	2;	ADD POSSIBLE CARRY TO PRODUCT HIGH
4021	85 61	STA	$61	3;	BYTE
4023	CA	DEX		2;	DECREMENT BIT COUNT
4024	D0 E2	BNE	$4008	2;	DONE? IF NOT, LOOP
4026	60	RTS		6;	
				392;	TOTAL MACHINE CYCLES REQUIRED

The Transfer of Results Routine does the following:

1. transfers the 16-bit product from PE to locations $1007 and $1008 in SM

2. transfers the 16-bit product from PE to locations $1001 and $1002 in SM

3. transfers the 16-bit product from PE to locations $1003 and $1004 in SM

4. transfers the 16-bit product from PE to locations $1005 and $1006 in SM.

8 x 8-BIT MULTIPLICATION TRANSFER OF RESULTS ROUTINE

LOCATION	OBJECT CODE	MNEMONIC	OPERAND	N	COMMENT
3F00	A5 60	LDA	$ 60	3;	GET PRODUCT LOW BYTE FROM PE
3F02	8D 07 10	STA	$1007	4;	TRANSFER TO SM
3F05	A5 61	LDA	$ 61	3;	GET PRODUCT HIGH BYTE FROM PE
3F07	8D 08 10	STA	$1008	4;	TRANSFER TO SM
3F0A	8D 00 C1	STA	$C100	4;	SET CP=1
3F0D	A5 60	LDA	$ 60	3;	GET PRODUCT LOW BYTE FROM PE
3F0F	8D 01 10	STA	$1001	4;	TRANSFER TO SM
3F12	A5 61	LDA	$ 61	3;	GET PRODUCT HIGH BYTE FROM PE
3F14	8D 02 10	STA	$1002	4;	TRANSFER TO SM
3F17	8D 00 C3	STA	$C300	4;	SET CP=2
3F1A	A5 60	LDA	$ 60	3;	GET PRODUCT LOW BYTE FROM PE
3F1C	8D 03 10	STA	$1003	4;	TRANSFER TO SM
3F1F	A5 61	LDA	$ 61	3;	GET PRODUCT HIGH BYTE FROM PE
3F21	8D 04 10	STA	$1004	4;	TRANSFER TO SM
3F24	8D 00 C5	STA	$C500	4;	SET CP=3
3F27	A5 60	LDA	$ 60	3;	GET PRODUCT LOW BYTE FROM PE
3F29	8D 05 10	STA	$1005	4;	TRANSFER TO SM
3F2C	A5 61	LDA	$ 61	3;	GET PRODUCT HIGH BYTE FROM PE
3F2E	8D 06 10	STA	$1006	4;	TRANSFER TO SM
3F31	8D 00 C7	STA	$C700	4;	SET CP=0
3F34	60	RTS		6;	RETURN FROM SUBROUTINE
				78;	TOTAL MACHINE CYCLES REQUIRED

Since the initialization routine disables all but the CP, it is necessary to know which PE is the CP before initialization. The initialization routine presented previously assumes that PE is the CP prior to initialization and will not work if this is not the case. If one desires that the CP be a PE other than PE , the software must be modified. The multiplication and transfer of results routines do not require that PE be the CP.